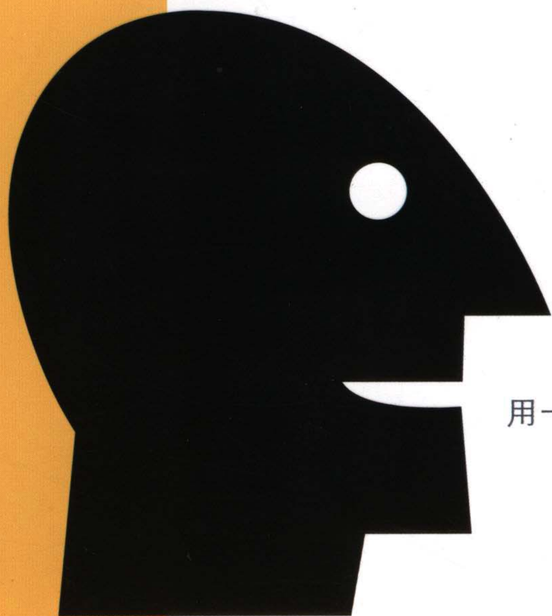
计算机科学先进技术译丛

MANNING

JavaScript开发实战

Get Programming with JavaScript



（英）约翰·拉尔森（John R. Larsen）著
孙晓青 周伟 译

交互式的学习方法

用一个完整的游戏编程项目贯穿全书

提供方便的在线编程环境

在线提供源代码





机械工业出版社
CHINA MACHINE PRESS

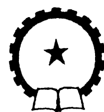
VISIT…

LANZAROTE Caliente.COM

计算机科学先进技术译丛

JavaScript 开发实战

[英]　约翰·拉尔森（John R. Larsen）　著
孙晓青　周　伟　译



机 械 工 业 出 版 社

本书是一本 JavaScript 开发入门指导书，主要介绍了变量、对象、函数、数组、构造函数、方括号运算符、作用域、条件语句、模块、模型、视图、控制器、构建网页、控件、模板、数据加载等内容。书中涉及的概念都附有简短示例。此外，还有一个贯穿全书的持续示例——冒险游戏 The Crypt。读者可以在一个交互式网站上直接运行书中的代码。

Original English language edition published by Manning Publications, USA. Copyright©2016 by Manning Publications, Simplified Chinese–language edition copyright © 2017 by China Machine Press. All rights reserved.

This title is published in China by China Machine Press with license from Manning Publications. This edition is authorized for sale in China only, excluding Hong Kong SAR, Macao SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书由Manning Publications授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）出版与发行。未经许可的出口，视为违反著作权法，将受法律制裁。

北京市版权局著作权合同登记 图字：01-2016-8234 号。

图书在版编目（CIP）数据

JavaScript 开发实战 /（英）约翰 • 拉尔森（John R. Larsen）著；孙晓青，周伟译.
—北京：机械工业出版社，2018.1
（计算机科学先进技术译丛）
书名原文：Get Programming with JavaScript
ISBN 978-7-111-58671-5

Ⅰ. ①J… Ⅱ. ①约… ②孙… ③周… Ⅲ. ①JAVA 语言－程序设计
Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 300956 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
责任编辑：车　忱
责任校对：张艳霞
责任印制：孙　炜
保定市中画美凯印刷有限公司印刷
2018 年 3 月第 1 版 • 第 1 次印刷
184mm×260mm • 24.75 印张 • 576 千字
0001－3000册
标准书号：ISBN 978-7-111-58671-5
定价：89.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
服务咨询热线：（010）88361066
读者购书热线：（010）68326294
（010）88379203

网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
教育服务网：www.cmpedu.com
金 书 网：www.golden-book.com

封面无防伪标均为盗版

The Crypt：一个持续示例

本书从始至终一直在持续开发一个名为 The Crypt 的基于文本的冒险游戏。在这个游戏中，玩家将在地图上进行探索，从一个地方移动到另一个地方，得到能够帮助他们解决问题和跨越障碍的有用物品。在每章的最后一节，会使用在该章所学习到的新内容对该游戏进行更新和完善。读者可以看到如何使用每章的编程知识编写小程序，然后将这些小程序组合生成一个大程序。

游戏元素	任　　务	JavaScript	章　节
玩家（Players）	确定我们所需要的每位玩家的信息	变量（Variables）	第 2 章
	收集在某一个地点中玩家的信息	对象（Objects）	第 3 章
	显示玩家的信息在控制台上	函数（Functions）	第 4–7 章
	创建每个玩家收集到的物品列表	数组（Arrays）	第 8 章
	组织创建玩家的代码	构造函数（Constructors）	第 9 章
地点（Places）	创建结构类似的探索地点	构造函数（Constructors）	第 9 章
	将地点与出口相连接	方括号运算符（Square bracket notation）	第 10 章
动作（Game）	添加简单的功能：移动、收集物品和显示信息	方括号运算符（Square bracket notation）	第 10 章
地图（Maps）	将地点与出口相连接	方括号运算符（Square bracket notation）	第 10 章

译者序

相比于市场上其他 JavaScript 编程书籍，本书的精彩之处在于将 JavaScript 理论与编程实践完美结合。这得益于本书作者既是一位资深教师又具有丰富的编程经验。他所秉持的教学理念是在实战中而非书本中学习编程。在本书中，该理念体现为没有繁杂的术语和晦涩的理论，取而代之的是简洁明快的语言和轻松诙谐的示例。正是在这些示例的一步步更新和完善的过程中，读者可以切身体会 JavaScript 编程的乐趣以及油然而生的成就感。

本书的另外一个亮点在于所有代码都可以在免费的 JS Bin 沙盒中运行，在这个允许犯错的安全空间里，读者可以亲自创建、扩展、调试和修改代码，这种真枪实弹的编程体验一定比纸上谈兵的学习方法更加高效和有趣，对编程学习者而言是不可多得的好机会。

读者也可访问 www.golden-book.com 或访问原书网址（见导读）下载随书代码。

总之，这是一本能够让读者轻松愉快地掌握 JavaScript 基础知识，并不断提高编程技能的书，同时这本书还为读者深入探索 JavaScript 指明了方向。

本书主要由上海工程技术大学的孙晓青和周伟翻译。徐晶晶、孙家璐、余泓臻、杨华、易显尊也参与了本书的翻译工作，在此一并表示感谢！

由于水平有限，不妥之处，恳请读者指正，邮件地址：cntysxq@163.com。

序言

本书作者 John 在电子邮件中邀请我为《JavaScript 开发实战》一书撰写序言。浏览此书后，我必须承认，最让我动心之处在于这本书一直在使用我的 JS Bin，而且一直鼓励读者在 JS Bin 中运行书中代码。JS Bin 始创于 2008 年，为程序员合作开发软件所用，他们可以在 JS Bin 中测试、调试、实验和共享程序。JS Bin 也适用于教育领域，John 的教师背景和他的编程实践相结合，恰恰与 JS Bin 理论联系实践的理念完美契合。我坚信，编码实践是学习编程的最佳方法，在 JS Bin 这个允许犯错的安全空间里，鼓励学习者创建、扩展、调试和修改代码，这对于初学者学习编程是不可多得的好机会。

JS Bin 是一个用 JavaScript 编写的应用程序。作为其开发者，我很高兴看到 John 在本书中采用它来教授初学者。当然，每个人会采用各自不同的方法学习编程，所花费的时间也会长短各异，但是，实践的方法无疑会让所有的学习者受益。JS Bin 为编程者提供了一个可实时访问的学习环境，作为免费资源，它不需要安装。本书为 JS Bin 新手起步提供入门指导，为读者编程实践提供理论支持，还为用户游戏探险提供用武之地。

记得在十多年前，当我看到对象的圆点运算符时，心中不解，就到 Google 上去搜索其义。如果我当时能够读到 John 编写的这本 JavaScript 入门书籍，就不会历经那么多次的搜索失败！这本书并没有包罗万象，但确实明晰了 JavaScript 的核心概念，并贴心为读者考虑，鼓励他们扩展 JavaScript 语言知识并提高编程技能。本书不仅示例丰富，而且还提供很大的空间供读者继续深入和提高。贯穿本书的游戏 The Crypt 会手把手地教会读者如何构建一个有实际用途的应用程序。

我有幸为程序员同行们创建了一些编程工具，特别是 JavaScript 工具。我们在编程中体会生活乐趣，收获经济利益，同时实现人生价值。欢迎读者加入我们的行列！谁又能知道这些新生力量会为我们带来怎样的伟大作品和伟大想法？但即使是小作品和小想法又何妨！我很高兴看到你们这令人兴奋的第一步将从 JS Bin 迈出。欢迎大家来 JS Bin 创建代码、修改代码、共享和构建代码模块库！《JavaScript 开发实战》将向读者展示如何管理代码库，如何一步步组合构建更大的项目（在 JS Bin 上甚至还可以使用 HTML 和 CSS 面板）。

亲爱的读者，希望你也喜欢这本书。我期望你在学完本书之后，能牢固地掌握 JavaScript 编程技能。

Remy Sharp

JS Bin 创始人

前言

我的编程生涯始于 1982 年。那一年，我开始使用 BASIC 语言在内存仅有 3.5KB 的 Commodore VIC-20 计算机上进行编程。当时，我需要将一本杂志上的指令逐行输入计算机，这个过程相当耗时并容易出错，但这确实培养并提高了我对细节的高度关注。那时，根本没有复制和粘贴，因此只能逐行阅读和输入指令。尽管如此艰难，程序最终还是从纸上转换到了计算机的内存里。那种激动的感觉仍然历历在目，我的编程生涯从此启航。

我一边紧盯着代码，一边仔细地思考，试图理解一条条指令并努力跟上程序跳跃的流程。并非所有的代码都清楚明白、一目了然，有时将一个程序压缩进 3.5KB 内存里确实需要一些编程技巧。但是，一点一点，一步一步，我最终还是解开了程序的秘密。有时候，程序会因为我的打字错误而无法运行；有时候，程序会因为代码本身有错误而无法运行。我虽然不能保证每一次都成功，但是大多数时候，程序最终还是得以运行了。

其实，程序最终变成垃圾的情况也时有发生！有时候，我一伸手不小心碰到了 VIC-20 计算机的开关，程序就永远消失了（其实，花费五分钟时间用一盒磁带就可以保存程序，但是有些程序似乎并不值得这么做）。我通常并不懊恼，也不认为这是在浪费时间。因为我已经看到文本转换成电脑屏幕上一个个可运转的程序（即使是垃圾），这已经足够让我惊喜不已。

如今，基于计算机编程的智能手机、平板电脑、无人机和阿尔法狗接连不断地带给我们惊喜。计算机编程简直就像是魔术师和变形金刚，把我们输入的一行行指令变为一个网页、一个游戏或是一个应用，这个过程太让我着迷了！

我在英国当老师，有幸能够教 16～17 岁的孩子编程。我的教学信条是让孩子们从第一课就开始动手编程：输入代码，并尽快看到结果。我希望他们自始至终都保持好奇，并乐意做实验。看到他们在代码中历险，并将想象变成现实，脸上呈现出甜美笑容时，我感觉非常幸福！正因为我们拥有像 JS Bin 这样的在线代码编辑环境，所以才能够让孩子们立刻尝试自己的想法并逐步构建一个项目。他们并不需要先学习一大堆语言知识就可以开始动手实践，恰恰是在这一次次练习和实验的过程中，他们会掌握一些概念。孩子们的编程技能与日俱增，课程开始时那些如同诡异象形文字般的代码到课程结束时已经逐渐成为自然而然的计算机语言。能够参与并体会孩子们的学习过程，我感到实在是太美妙了！

我在当老师的同时，也在当程序员，开发教育应用程序来组织、分享和订购课程资源，创建在线手册，制作教案，管理进度表并编制测验。看到人们在日常工作中使用这些应用程序，我感觉很欣慰。我很幸运能够了解这些程序的目标受众，看到这些应用程序得以使用如此之久，我感到备受鼓舞！

我曾经为曼宁出版物有限公司审查过一些书稿，这些书稿的作者将我描述为程序员和老师，曼宁建议我自己也编写一本书。《JavaScript 开发实战》是我将编程教学实践转换为编程教学指导书的一次尝试。该书内容不仅包含代码清单，而且还有很多练习和在线补充材料，帮助读者逐渐理解概念。希望这本书能激发你的想象力，鼓舞你的编程探险之旅。祝你好运，并乐在其中！

致谢

感谢曼宁出版物有限公司的 Robin de Jongh 建议我写这本书。感谢本书的编辑 Helen Stergius 在我写作过程中给予的耐心支持和建议。也感谢审阅这本书的 Philip Arny、Dr. Markus Beckmann、Rocio Chongtay、Sonya Corcoran、 Philip Cusack、 Alvin Raj、Conor Redmond、Ivan Rubelj、Craig Sharkie 和 Giselle Stidston，他们的反馈使本书日臻完善。特别感谢 Ozren Harlovic、Chuck Henderson、Al Sherer、Brian Hanafee 和 Romin Irani，他们对本书的诸多细节提供了建设性意见。

还要感谢 JS Bin 的创始人 Remy Sharp，他不仅积极回应我的各种问题和要求，而且还为本书作序。当然还要感谢他创建了 JS Bin！

最后，我要感谢所有为这本书的面世辛苦付出的曼宁工作人员：出版人 Marjan Bace 和编辑发行团队中的每一位工作人员，包括 Janet Vail、Mary Piergies、Tiffany Taylor、Linda Recktenwald、Dennis Dalinnik、Elizabeth Martin、Bonnie Culverhouse，以及许多其他幕后英雄。

导读

《JavaScript 开发实战》是一本入门指导书，适合没有任何编程经验的初学者。书中的代码都可以在 JS Bin 网站上在线运行。JS Bin 沙盒的妙处在于，使用者可以在其中进行代码试验而且可以立即查看结果。JS Bin 无需安装，读者只需接入互联网，便可以直接使用。如果没有接入互联网，也不必担心，本书中已经列出打印好的代码清单，并包含读者易懂的代码注释和文字解释。

本书所涉及的概念都附有简短示例。除此之外，还有一个贯穿全书的持续示例——The Crypt，它是一个基于文本的冒险游戏。

本书的读者对象

如果你在工作中经常使用电脑，使用各种应用程序，只是以前没有编写过程序，但是你又希望学习如何编写程序，那么这本书正适合你。本书没有对 JavaScript 进行面面俱到的讲述，而是通过大量的实例和练习帮助读者学习 JavaScript 编程，本书还鼓励读者自己去进一步思考和探索。本书同样适用于那些已经是程序员，目前正在寻找一本完整的 JavaScript 参考书的人群。如果你想要一本仔细详实的 JavaScript 入门书，也请驻足，对本书中基础知识的牢固掌握将有助于你未来轻松阅读其他编程图书。

本书的章节简介

本书内容一共 25 章，其中纸质印刷版包括 21 章内容，另外 4 章内容在出版商的网站上，可以在线获取：www.manning.com/books/get-programming-with-javascript。本书中有大量的代码清单和练习，以及难度渐进的示例，因此建议读者按顺序阅读，并且在线试验这些示例和练习，以期读者最终能够真正理解相关概念。

第一部分介绍 JavaScript 编程的一些核心概念。编程环境为 JS Bin 基于文本的控制台面板，该环境可以使读者专注于 JavaScript 而不会因为网页问题和 HTML 问题而分散注意力。其中：

- 第 1 章介绍编程的概念。本章的编程是指 JavaScript 编程在引入 JS Bin（一个可以立即编程的网站）沙盒环境之前的编程概念。本章还向读者介绍 The Crypt，这是一款基于文本的冒险游戏，它将一直贯穿本书始终。
- 第 2 章介绍变量，它是程序中标记和使用值的方法。变量犹如放着各类数据的碗，它可以存放数字数据或文本数据。变量的命名必须遵循某些规则。
- 在第 3 章中，读者将了解如何将值放入到各个对象中。对象犹如一个急救包，医护人员通常将急救包作为一个整体来携带和传递，只有在需要时才会将急救包内的某个物品单独取出来。我们可以把 JavaScript 对象看作一个单独的项目，只有在需要时才会访问对象的某个属性。

- 本书的第 4～7 章介绍函数。函数是 JavaScript 的核心内容，使用函数可以帮助程序员组织代码，并可以根据需要多次执行一套指令集合。本书用 4 个章节的篇幅来讲述函数，以便读者能够熟练掌握如何定义和使用函数，如何向函数传递数据，以及如何将函数与对象完美结合。
- 第 8 章介绍如何创建有序列表，即数组。在编程中，程序员经常采用列表来存放博客帖子、日程活动、各个用户名、影评等一系列事项。在本章，读者将学习如何创建列表、访问列表、操作列表以及删除列表。
- 对象是 JavaScript 的核心，并且编程中会经常创建许多对象。例如，一个日历可以包含数千个事件，一个冒险游戏可以包含数十个地点。构造函数是一种对许多相似的对象进行简化创建的方式。第 9 章探讨构造函数在何时可用，以及如何定义和使用构造函数。
- 在第 10 章，读者会看到方括号运算符，使用方括号运算符可以访问存储在 JavaScript 对象中的值。作为一种替代性访问对象的方法，采用方括号运算符可以更灵活地获取和设置对象的属性。利用方括号运算符，读者可以编写一些小程序来处理不可预测的值，不可预测性可能来自外部数据或用户的输入。

本书在第一部分介绍 JavaScript 的核心概念之后，将在第二部分介绍如何更好地组织代码。

- 第 11 章讨论全局变量和局部变量。在整个程序中都可以看到使用全局变量的危险，也可以看到在函数内部定义的局部变量的优势。在决定采用全局变量还是局部变量的过程中，程序员应该考虑哪些人有权使用程序的代码，还应该考虑如何区分程序的接口和实现。
- 想要在程序中使用条件，请学习本书第 12 章内容。条件意味着只有在满足某条件的情况下才能执行代码，这样会增加程序的灵活性，并允许程序先检查用户的输入是否满足相关条件，然后再使用用户的输入。
- 随着程序越来越长，有必要将程序分割成模块。将那些可以组合和替换的文件独立分割开成为模块，可以使模块更加灵活好用，方便移植到各个项目中，还可以使每个模块的功能更加独立，易于更新和维护。第 13 章探讨如何将程序代码进行模块化处理，其中包括两个重要概念：命名空间和立即调用函数表达式。
- 在读者学会创建模块的技巧之后，在第 14～16 章中可以看到模块的三个重要应用：模型帮助程序员使用数据（例如日历事件、博客帖子或影评）；视图将数据呈现给用户（例如文本、HTML 或图形）；控制器使用模型和视图，对用户操作做出响应，然后对模型进行更新，再传递到视图进行显示。

本书第三部分介绍如何使用 JavaScript 更新网页，并通过按钮、下拉列表和文本框对用户的输入做出响应。这部分还介绍了用于显示重复的动态数据的模板，以及如何将此类数据加载到现有页面。

- 第 17 章简要介绍超文本标记语言（HTML），该语言用于指定网页中内容的结构（如标题、段落或列表项），以及图像、视频、脚本和样式表等其他资源的加载方式。本章还介绍如何使用 JavaScript 访问和更新页面的内容。
- 为了捕获用户输入，需要使用 HTML 控件，如按钮、下拉列表和文本框。第 18 章介

绍如何设置代码才能使用户输入信息，并且在用户单击某个按钮时，程序就会运行。

- 模板提供了一种使用占位符来设计数据呈现的方法。在第 19 章，读者将学习如何在页面中使用 HTML 模板，并用数据来替换模板中的占位符。这样可以避免将 JavaScript、HTML 和数据混在一起，能够用一种清晰明了的方式，使用格式化信息来构建网页。
- 第 20 章讲解如何使用 XMLHttpRequest 对象将更多数据加载到网页中，通常称为 Ajax。这些技术能够针对用户的动作，使用新数据更新部分页面，从而使应用程序更加具有即时响应性。
- 第 21 章为本书最后一章，讨论文本编辑器、集成开发环境以及在不用 JS Bin 的情形下如何组织自己的文件。本章还为读者进一步深入学习 JavaScript 提出建议，并预祝读者在编程的历险中顺利前行。

本书的第 22～25 章内容在纸版图书中没有包括，仅提供在线版本，请读者访问 www.manning.com/books/get-programmingwith-javascript。这 4 章为进阶内容，介绍如何使用 Node.js 和 Express.js 在服务器上编程，如何使用 XHR 轮询服务器，并与 Socket.IO 进行实时通信。

关于本书中的代码

本书包含许多源代码示例，以编号代码清单和普通文本形式呈现。源代码使用等宽字体区别于普通文本。有些代码用粗体突出显示。

在多数情况下，本书对原始源代码的格式进行了重新编排，添加了换行和缩进以适合本书的页面空间。在少数情况下，如果这样还不能解决问题，本书就在代码列表中采用了行延续标记（➥）。此外，源代码中的注释大多被删除，因为本书在文本部分已经对相关代码进行了描述。在本书中，代码清单、重要概念与代码注释相伴而行。

本书中的大多数代码清单都包含一个指向链接，通过该链接可以在 JS Bin 网站运行这些代码并进行实验。另外，读者还可以在 GitHub 和曼宁网站获取代码，链接如下：

https://github.com/jrlarsen/getprogramming

www.manning.com/books/get-programming-with-javascript

其他在线资源

本书的网站提供本书的练习答案、视频教程、进阶文章、JavaScript 学习指南，以及其他互联网上的资源链接。

本书的网站链接是 www.room51.co.uk/books/getprogramming/index.html。

关于本书的作者

John Larsen 是一位对教育研究感兴趣的数学和计算机教师，他拥有数学和信息技术两个硕士学位。John Larsen 从 1982 年开始编程，在 1993 年编写简单的数学教学程序，在 2001 年建立多个网站，在 2006 年开发了多个数据驱动的基于网络的教学应用程序。

作者在线论坛

购买《JavaScript 开发实战》一书后可以免费访问由曼宁出版物有限公司运营的客户网络

论坛。读者可以对该书进行评论，提出技术问题，并从本书作者和其他用户处获得帮助。欲访问并参与该论坛，请访问 www.manning.com/books/get-programming-with-javascript。该论坛一经注册，便可得到在线帮助。曼宁出版物有限公司承诺为读者提供这样一个交流平台，在这个论坛中，读者之间以及读者和作者之间都可以进行有意义的对话。我们不会对作者在该论坛中的具体参与量做出任何承诺，作者在该论坛上与读者的互动都是出于自愿并且无任何报酬的。希望读者能够向本书作者提出一些具有挑战性的问题，以免他在该论坛的兴趣和热情丧失殆尽！

在本书出版发行期间，读者可以从出版商的网站访问作者的在线论坛，并浏览该论坛的历史记录。

速览

第一部分　控制台上的核心概念　/ 1

第 1 章　编程、JavaScript 和 JS Bin / 3
第 2 章　变量：在程序中存储数据　/ 13
第 3 章　对象：数据分组　/ 22
第 4 章　函数：按需执行代码　/ 34
第 5 章　参数：将数据传递给函数　/ 49
第 6 章　返回值：从函数获取数据　/ 60
第 7 章　对象参数：将对象传递给函数　/ 72
第 8 章　数组：将数据存入列表　/ 91
第 9 章　构造函数：构建带有函数的对象　/ 107
第 10 章　方括号运算符：灵活的属性名称　/ 130

第二部分　组织代码　/ 149

第 11 章　作用域：隐藏信息　/ 151
第 12 章　条件：有选择地运行代码　/ 175
第 13 章　模块：分解程序　/ 196
第 14 章　模型：使用数据　/ 220
第 15 章　视图：显示数据　/ 234
第 16 章　控制器：连接模型和视图　/ 248

第三部分　浏览器中的 JavaScript / 265

第 17 章　HTML：构建网页　/ 267
第 18 章　控件：获取用户输入　/ 286
第 19 章　模板：使用数据填充占位符　/ 304
第 20 章　XHR：加载数据　/ 326
第 21 章　结语：使用 JavaScript 编程　/ 345

在线部分（资源网址：www.manning.com/books/get-programming-with-javascript）

第 22 章　Node：在浏览器之外运行 JavaScript
第 23 章　Express：构建 API
第 24 章　轮询：用 XHR 重复请求
第 25 章　Socket.IO：实时通信

目录

译者序
序言
前言
致谢
导读
速览

第一部分　控制台上的核心概念

第 1 章　编程、JavaScript 和 JS Bin …… 3
1.1　编程 …… 3
1.2　JavaScript …… 4
1.3　在实践和思考中学习 …… 4
1.4　JS Bin …… 5
1.4.1　JS Bin 面板 …… 5
1.4.2　在 JS Bin 上运行代码清单 …… 6
1.4.3　输出到控制台 …… 7
1.4.4　代码注释 …… 7
1.4.5　Further Adventures——进阶练习 …… 8
1.4.6　错误消息 …… 8
1.4.7　行号 …… 9
1.4.8　获取账户 …… 9
1.5　游戏 The Crypt——本书的一个持续示例 …… 9
1.5.1　运行 The Crypt …… 10
1.5.2　创建 The Crypt 的具体步骤 …… 10
1.6　更多示例和练习 …… 12
1.7　浏览器的兼容性 …… 12
1.8　本章小结 …… 12
第 2 章　变量：在程序中存储数据 …… 13
2.1　什么是变量 …… 13
2.2　变量的声明和赋值 …… 13
2.2.1　变量声明 …… 14
2.2.2　变量赋值 …… 14
2.2.3　一步实现变量声明和赋值 …… 16
2.2.4　先运算再赋值 …… 17

2.3 选择合适的变量名 …… 18
2.3.1 关键字和保留字 …… 18
2.3.2 变量的命名规则 …… 19
2.3.3 骆驼式命名法（camelCase） …… 19
2.3.4 使用描述性变量名 …… 19
2.4 The Crypt——玩家变量 …… 20
2.5 本章小结 …… 21
第 3 章 对象：数据分组 …… 22
3.1 变量需要分组 …… 22
3.2 创建对象 …… 23
3.2.1 创建一个空对象 …… 24
3.2.2 属性“键–值对” …… 24
3.3 访问对象的属性 …… 26
3.4 更新对象的属性 …… 28
3.5 其他示例 …… 29
3.5.1 撰写一条博客 …… 29
3.5.2 创建一个日历 …… 29
3.5.3 天气怎么样？ …… 30
3.5.4 一个小测验 …… 31
3.5.5 创建自己的程序 …… 31
3.6 游戏 The Crypt——玩家对象 …… 31
3.7 本章小结 …… 33
第 4 章 函数：按需执行代码 …… 34
4.1 重复问题 …… 34
4.1.1 将对象的属性作为文本进行显示 …… 34
4.1.2 加税算出总成本 …… 36
4.2 定义函数和调用函数 …… 37
4.2.1 定义新函数 …… 37
4.2.2 函数表达式和函数声明 …… 38
4.2.3 使用函数 …… 39
4.2.4 使用函数的步骤 …… 40
4.3 减少重复 …… 40
4.3.1 函数可用于将对象的属性作为文本进行显示 …… 40
4.3.2 加税并显示总成本的函数 …… 43
4.4 使代码易于阅读和更新 …… 44
4.4.1 更新 showMovieInfo 函数 …… 45
4.5 游戏 The Crypt——显示玩家的信息 …… 46
4.5.1 用函数显示玩家的信息 …… 47
4.6 本章小结 …… 48

第 5 章　参数：将数据传递给函数 49
5.1　函数重用 49
5.2　将信息传递给函数 50
5.2.1　将实参传递给函数 50
5.2.2　将多个实参传递给一个函数 53
5.3　The Crypt——显示玩家信息 54
5.3.1　显示玩家的姓名 55
5.3.2　显示玩家的健康值 56
5.3.3　显示玩家的位置 57
5.3.4　合并显示玩家的信息 58
5.4　本章小结 59
第 6 章　返回值：从函数获取数据 60
6.1　从函数返回数据 60
6.1.1　用返回值替代函数调用 60
6.1.2　关键字 return 61
6.1.3　使用参数来确定返回值 62
6.2　在控制台提示符下进行实验 64
6.2.1　调用函数 64
6.2.2　声明新的变量 65
6.3　The Crypt——构建玩家信息字符串 66
6.3.1　为玩家的姓名、健康值和位置构建字符串 67
6.3.2　用一个函数显示玩家的信息——把几个函数集合在一起 68
6.4　本章小结 70
第 7 章　对象参数：将对象传递给函数 72
7.1　使用对象作为参数 72
7.1.1　访问对象参数的属性 73
7.1.2　给对象参数添加属性 74
7.2　从函数返回对象 75
7.2.1　构建行星——对象创建函数 75
7.2.2　二维空间的点 77
7.3　方法——设置函数作为对象的属性 78
7.3.1　命名空间——将相关函数组织到一起 79
7.3.2　Math 方法 80
7.3.3　String 方法 81
7.3.4　spacer——将更多的方法收入命名空间 83
7.3.5　进一步探索命名空间 85
7.4　The Crypt——将玩家对象作为参数 88
7.5　本章小结 89

第 8 章　数组：将数据存入列表 …… 91
8.1　创建数组并访问元素 …… 91
8.1.1　创建数组 …… 91
8.1.2　访问数组元素 …… 93
8.2　数组方法 …… 96
8.2.1　添加和删除元素 …… 96
8.2.2　截取和拼接数组 …… 97
8.2.3　使用 forEach 访问每一个元素 …… 98
8.3　The Crypt——玩家的物品数组 …… 103
8.4　本章小结 …… 105
第 9 章　构造函数：构建带有函数的对象 …… 107
9.1　使用函数构建对象 …… 108
9.1.1　添加属性 …… 108
9.1.2　添加方法 …… 109
9.2　使用构造函数构建对象 …… 111
9.2.1　构造函数 …… 111
9.2.2　使用 Planet 构造函数创建一个新世界 …… 114
9.2.3　使用 instanceof 运算符区分对象 …… 115
9.3　建造大师——两个构造函数的示例 …… 115
9.4　The Crypt——为玩家提供掠夺地 …… 118
9.4.1　构建 Place 构造函数——名称标题和描述 …… 119
9.4.2　构建 Place 构造函数——囤积物品 …… 120
9.4.3　构建 Place 构造函数——探索出口 …… 121
9.5　The Crypt——简化玩家创建代码 …… 123
9.5.1　整理玩家属性 …… 124
9.5.2　将函数转换为方法 …… 126
9.5.3　为玩家分配位置 …… 127
9.5.4　使用 null 作为对象的占位符 …… 127
9.6　本章小结 …… 128
第 10 章　方括号运算符：灵活的属性名称 …… 130
10.1　用方括号运算符替代圆点运算符 …… 130
10.1.1　使用方括号——人的姓名作为键 …… 131
10.1.2　最大限度地利用方括号运算符——单词统计 …… 134
10.2　The Crypt——使游戏出口更加刺激好玩 …… 137
10.2.1　使用对象存放出口 …… 138
10.2.2　创建一个添加并显示出口的函数 …… 139
10.2.3　设置每个场所对象的出口集合 …… 140
10.2.4　将 exits 对象添加到完整的 Place 构造函数 …… 142
10.2.5　测试 Place 构造函数 …… 144

10.3 The Crypt——开始游戏！ 145
10.3.1 刷新显示——render 146
10.3.2 探索地图——go 146
10.3.3 收集物品——get 147
10.3.4 设计一个更大的冒险游戏——Jahver 的船 147
10.4 下一步目标 147
10.5 本章小结 148

第二部分 组 织 代 码

第 11 章 作用域：隐藏信息 151
11.1 全局变量的危险性 152
11.1.1 访问所有区域——偷窥和篡改 152
11.1.2 访问所有区域——借助于实现 153
11.1.3 命名冲突 155
11.1.4 难以查找的错误 155
11.2 局部变量的优势 155
11.3 接口——控制访问权限并提供可用功能 157
11.3.1 使用一个函数来隐藏变量 158
11.3.2 使用 getCount 创建多个独立计数器 159
11.3.3 用构造函数创建多个独立的计数器 160
11.4 创建一个简单的测验应用程序 161
11.4.1 将对象用作一个命名空间 162
11.4.2 隐藏问题数组 163
11.5 The Crypt——隐藏玩家信息 165
11.5.1 当前的 Player 构造函数——全部内容都公开 165
11.5.2 更新版的 Player 构造函数——某些变量被隐藏 166
11.6 The Crypt——隐藏地点信息 168
11.7 The Crypt——用户交互 170
11.7.1 接口——go 和 get 172
11.7.2 隐藏程序的实现 172
11.8 本章小结 173
第 12 章 条件：有选择地运行代码 175
12.1 有条件地执行代码 175
12.1.1 严格相等运算符 176
12.1.2 if 语句 176
12.1.3 else 子句 177
12.1.4 隐藏函数中的密码数字 178
12.2 利用 Math.random()生成随机数 180
12.3 使用 else if 创造更多条件 182
12.3.1 比较运算符 184

12.4 在测验应用程序中检查答案 185
12.4.1 变量的便捷声明方式：使用 var 关键字声明多个变量 187
12.4.2 显示测验中的问题 188
12.4.3 移至下一道题目 188
12.4.4 检查用户的答案 189
12.4.5 处理用户的答案 189
12.4.6 返回接口对象 189
12.5 The Crypt——检查用户的输入 190
12.5.1 go 方法解读 190
12.5.2 永远不能轻信用户输入的信息 191
12.5.3 确保探索不会中断——使用 if 语句 192
12.6 本章小结 194
第 13 章 模块：分解程序 196
13.1 了解 JS Bin 上的 bin 和文件 197
13.1.1 创建一个 bin 199
13.1.2 编写代码 199
13.1.3 记录文件名 199
13.1.4 查看单个代码文件 200
13.2 将文件导入其他项目 200
13.2.1 创建一个 bin 200
13.2.2 编写代码 200
13.2.3 添加 script 元素 201
13.2.4 刷新页面 202
13.2.5 运行程序 202
13.3 导入数字生成器——进阶示例 203
13.3.1 在测验应用程序中随机选择问题 203
13.3.2 在猜数字游戏中使用 between 函数 204
13.4 导入多个文件 205
13.5 冲突——新导入的代码会覆盖原先的变量 207
13.5.1 变量冲突 209
13.5.2 通过使用命名空间减少冲突 210
13.6 立即调用函数表达式（IIFE） 211
13.6.1 识别函数表达式 213
13.6.2 调用函数 213
13.6.3 立即调用函数表达式 213
13.6.4 从 IIFE 返回信息 214
13.7 The Crypt——将代码分解成模块 215
13.7.1 跨模块共享命名空间 216
13.8 本章小结 219

第 14 章　模型：使用数据 220
14.1　构建健身应用程序——数据和模型 220
14.1.1　定义 User 构造函数 221
14.1.2　体会把数据作为 JavaScript 对象的感觉 222
14.1.3　将数据转换为用户模型 223
14.1.4　下一步目标 224
14.2　The Crypt——从游戏中分离出地图数据 224
14.2.1　地图数据 226
14.2.2　向 The Crypt 地图数据添加挑战 227
14.2.3　更新 Place 构造函数来应对挑战 228
14.2.4　使用地图数据构建游戏地图 229
14.2.5　运行游戏 232
14.3　本章小结 233
第 15 章　视图：显示数据 234
15.1　健身应用程序——显示最新的用户数据 234
15.1.1　创建第一个健身应用视图 235
15.1.2　使用模块切换健身应用视图 236
15.1.3　下一个目标 237
15.2　The Crypt——把显示代码从 Player 和 Place 模型中分离出来 238
15.2.1　为玩家创建视图 238
15.2.2　创建场所视图 242
15.3　与玩家交互——消息视图 246
15.4　本章小结 247
第 16 章　控制器：连接模型和视图 248
16.1　构建一个健身应用控制器 248
16.1.1　控制器的作用 249
16.1.2　构建一个健身应用控制器 249
16.1.3　将各个部分组合在一起用于健身应用程序 250
16.1.4　下一个目标 251
16.2　游戏 The Crypt——添加控制器 251
16.2.1　控制器在游戏中的作用 252
16.2.2　走近控制器代码 253
16.3　The Crypt——控制器代码的结构 253
16.4　The Crypt——开始和终止游戏 254
16.4.1　游戏初始化 254
16.4.2　监测玩家的健康值 255
16.4.3　更新显示——使用视图模块的函数 255
16.5　The Crypt——发布命令和解决难题 256
16.5.1　使用 game.get 拾取物品 256

16.5.2 列出挑战的各个属性 ······ 257
16.5.3 使用 game.go 来移动位置 ······ 258
16.5.4 用 game.use 打败豹子 ······ 259
16.6 The Crypt——运行游戏 ······ 262
16.7 The Crypt——下一个目标 ······ 263
16.8 本章小结 ······ 263

第三部分　浏览器中的 JavaScript

第 17 章　HTML：构建网页 ······ 267
17.1 HTML、CSS、JavaScript——构建网页 ······ 267
17.1.1 加载图层 ······ 268
17.1.2 在 JS Bin 中加载图层 ······ 269
17.2 HTML——简略介绍 ······ 269
17.2.1 从空白页面开始 ······ 270
17.2.2 添加内容 ······ 270
17.2.3 标记列表 ······ 271
17.2.4 一些常见的 HTML 元素 ······ 272
17.3 使用 JavaScript 向 Web 页面添加内容 ······ 273
17.3.1 通过 id 获取元素 ······ 274
17.3.2 函数声明 ······ 275
17.3.3 什么？不能运行 JavaScript？ ······ 275
17.4 显示数组中的数据 ······ 275
17.5 游戏 The Crypt——用网页视图方式显示玩家和地点 ······ 278
17.5.1 更新玩家和场所视图模块——render 方法 ······ 279
17.5.2 更新玩家和场所视图模块——代码清单 ······ 280
17.5.3 使用 JavaScript 严格模式 ······ 281
17.5.4 加载模块并在 HTML 中添加占位符 ······ 281
17.5.5 添加少许 CSS ······ 283
17.5.6 开始玩游戏 ······ 283
17.5.7 准备消息视图 ······ 284
17.6 本章小结 ······ 284
第 18 章　控件：获取用户输入 ······ 286
18.1 使用按钮 ······ 286
18.1.1 为页面添加按钮 ······ 287
18.1.2 编写函数来更新问候语 ······ 287
18.1.3 监听单击 ······ 288
18.2 使用 select 元素选择一个选项 ······ 289
18.2.1 向页面中添加 select 元素 ······ 289
18.2.2 对电影进行评分的函数和调用该函数的按钮 ······ 291

18.3 使用文本框读取用户输入 292
18.3.1 在页面中添加文本框 293
18.3.2 添加无序列表来显示评论内容 293
18.3.3 获取对新元素的引用 293
18.3.4 更新 rateMovie 函数 294
18.3.5 用 CSS 设计样式 295
18.4 游戏 The Crypt——玩家通过文本框发号施令 295
18.4.1 向页面添加控件 297
18.4.2 将文本框内容映射到游戏命令 297
18.4.3 使用 split、join、pop 和 shift 发布命令 297
18.4.4 使用 switch 在各选项间做出决定 299
18.4.5 执行命令——监听按钮点击 300
18.4.6 进入游戏 301
18.5 本章小结 302
第 19 章 模板：使用数据填充占位符 304
19.1 构建一个新闻页面——重大新闻 304
19.1.1 对比新闻数据和 HTML 305
19.1.2 通过连接字符串来构造 HTML 306
19.1.3 使用 HTML 模板进行设计 306
19.1.4 使用 script 标签包裹模板 306
19.2 用一个字符串替换另一个字符串 307
19.2.1 连续调用 replace 308
19.3 while 循环——多次替换字符串 309
19.3.1 在满足条件时重复执行代码 311
19.3.2 while 循环 311
19.3.3 替换一个可以找到的字符串 312
19.3.4 使用正则表达式替换字符串 313
19.4 自动替换模板中的占位符 313
19.4.1 将模板占位符与对象属性相匹配 313
19.4.2 为每个“键”填充所有的占位符 314
19.4.3 使用模板构建多条新闻的列表 315
19.5 创建一个新闻页面——处理最新消息 316
19.5.1 创建模板模块和数据模块 316
19.5.2 导入模块 318
19.6 游戏 The Crypt——改善视图 319
19.6.1 为所有视图创建 HTML 模板 320
19.6.2 针对新模板更新视图 321
19.6.3 进入游戏 The Crypt 324
19.7 本章小结 324

第 20 章　XHR：加载数据 326
20.1　构建一个健身应用——检索用户数据 326
20.1.1　查找用户数据 327
20.1.2　加载用户数据——概要 328
20.1.3　加载用户数据——XMLHttpRequest 构造函数 329
20.1.4　加载用户数据——使用 JSON.parse 解析 XHR 响应 330
20.1.5　加载 JS Bin 数据——一个好用的函数 331
20.1.6　升级健身应用程序 332
20.1.7　健身应用程序——下一个目标 334
20.2　JSON——一种简单的数据格式 334
20.2.1　使用 JSON.parse 将 JSON 转换为对象和数组 335
20.3　游戏 The Crypt——根据需要加载地图 336
20.3.1　使用 JS Bin 文件代码指定出口 336
20.3.2　使用缓存——针对每个地点只加载一次 337
20.3.3　使用地图管理器替换地图数据模块和地图构建器模块 338
20.3.4　更新游戏控制器 340
20.3.5　构建游戏页面 341
20.3.6　进入游戏 The Crypt 343
20.4　本章小结 343
第 21 章　结语：使用 JavaScript 编程 345
21.1　在本地处理文件 345
21.1.1　编写代码 345
21.1.2　保存文件 346
21.1.3　在浏览器中打开页面 347
21.1.4　合并和压缩文件 348
21.2　获取帮助 349
21.3　后续推荐 349
21.3.1　本书网站 349
21.3.2　图书 349
21.3.3　网站 350
21.3.4　实践——熟能生巧 350
21.4　本章小结 350

在线部分（资源网址：www.manning.com/books/get-programming-with-javascript）

第 22 章　Node：在浏览器之外运行 JavaScript
第 23 章　Express：构建 API
第 24 章　轮询：用 XHR 重复请求
第 25 章　Socket.IO：实时通信
索引 351

第一部分　控制台上的核心概念

《JavaScript 开发实战》的第一部分讲解了程序设计中的一些核心概念。这些概念是后续代码的基础，将辅以示例逐一呈现。贯穿本书始末的重要主题是如何用变量来存储和读取值，如何用对象和数组来聚合值以及如何用函数来集合指令。在第一部分的结尾，将构建一个冒险游戏的开发版，名为 The Crypt。玩家可以探索地图上的各个场所，从一个房间移动到另一个房间，从一座坟墓移动到另一座坟墓，不断地收集宝贝。第 1 章通过介绍编程、JavaScript、JS Bin 以及玩家进行探险的在线代码环境来设置游戏的背景。让我们开始吧！ 1

第 1 章　编程、JavaScript 和 JS Bin

本章内容包括：

- 编程
- JavaScript
- JS Bin
- 本书持续范例：The Crypt

《JavaScript 开发实战》是一本入门级的编程书。通过代码示例，配合详细的讲解和视频辅导，本书将帮助读者提升编写代码的知识和技能。

本章首先介绍编程概念和 JavaScript 编程，然后介绍 JS Bin——读者即将学习使用的在线编程环境，最后是本书的一个持续范例：游戏 The Crypt。

1.1　编程

编程是给计算机发出一系列它可以理解的指令。程序无处不在，例如火星探测器、大型强子对撞机、发动机管理系统、金融市场、无人机、手机、平板电脑、电视、医学设备等，程序的威力令人叹为观止。程序短则只有几行代码，长则可达数百万行，由各个简单的模块组建成复杂的解决方案。

计算机的内部是二进制、计数器、寄存器、总线和存储器的天地。人们最初使用低级编程语言让它们运行工作，例如：机器语言和汇编语言。幸运的是，我们已经创建了更容易阅读、继承和使用的高级语言。我们编写的程序代码可以让任何人理解，下面是一些接近高级语言的伪代码：

```
increase score by 100
if score is greater than 5000 print "Congratulations! You win!"
otherwise load new level
```

不同的编程语言会条理清晰地告诉你该如何写这样的代码：一些编程语言会更多地使用符号，另一些编程语言会更多地使用自然语言。用 JavaScript 实现上述伪代码，如下所示：

```
score = score + 100;
if (score > 5000)
{
    alert("Congratulations! You win!");
}
else
{
```

```
        loadNewLevel();
    }
```

圆括号、花括号和分号都是编程语言语法的一部分，都是用来阐明程序代码的规范，可以被计算机所理解。以上程序将自动翻译成能够让计算机执行的低级语言代码。

在上述 JavaScript 代码段中，loadNewLevel()语句的作用是在游戏中加载新的级别。关于如何加载游戏的新级别，在程序的其他地方会有更多的代码描述来逐步说明。编程中有一种技巧是将复杂的项目分解成多个能够执行特定功能的小模块，然后将这些小模块连接起来，以实现主程序的需要。

编程语言有很多，如 Java、C、PHP、Python 和 Swift，下面看看为什么本书会选择 JavaScript。

1.2 JavaScript

JavaScript 是一种非常流行的程序设计语言，常用于网络浏览器，其实在其他环境中，JavaScript 也变得越来越受欢迎。JavaScript 为网页增加了互动性——从简单的动画效果到成熟网页应用程序的表单验证。服务器程序使得在互联网上可以使用的文件、网页和其他资源都能够通过带有 Node.js 的 JavaScript 编写出来。其他软件也可以使用 JavaScript 编写，例如 Photoshop 和 Minecraft，另外，一些数据库能够存储 JavaScript，同时也可以使用 JavaScript 进行数据查询。随着越来越多的网络化对象出现在物联网中，JavaScript 在传感器、机器人、无人机和 Arduino 式电子器件的编程领域越来越受欢迎。

4

学习程序设计将使你获得一项高明的技术，这些技术是通用的、实用的、有新意的、有创造性的、有趣的和有价值的，并且在这个时代也是有需求的。学习 JavaScript 编程会让你轻松驾驭世界上使用最广泛的计算机语言，开发出针对各种用途、设备、平台和操作系统的应用程序。

1.3 在实践和思考中学习

《JavaScript 开发实战》的学习理念是：通过在线沙盒中的编程练习，让程序开发者直接看到什么是可执行的，什么是不能执行的，通过不断地尝试挑战，读者将认真思考每一章的语法概念，这样的思考有助于读者更加深入地理解和学习。

在沙盒中运行程序可以立刻得到运行结果，有时候得到的结果会出乎意料，这将迫使程序开发者质疑自己的所思所知。他们有时能很快获得解决问题的思路和方法，有时却要花费很长的时间。这些问题的解决方案都需要开发者认真地思索，并通过实验去检验。好奇、努力和坚持是学习任何东西的关键态度，这些积极的学习态度一定会帮助你成为一名出色的程序员。

这并不是说学习编程会是一件苦差事！即使从事编程工作三十多年，我仍然觉得将代码转换成一个有用或是有趣的应用程序，将几行简单的语句结合起来就可以实现各种各样的效果是如此神奇和不可思议。看到别人因为使用我设计的程序而变得更有效率、更有条

理，哪怕仅仅是为他们带来更多的乐趣，也会让我感觉荣幸之至，编程让我如此快乐！

让我们做好准备，来一次探险之旅吧！如果刚开始就发现有些概念非常棘手，请不要灰心，慢慢来，多做一些练习。不要忘记使用本书网站上的资源，那里有代码清单、解决方案、学习视频和进阶学习，网址为www.room51.co.uk/books/getprogramming/index.html。学习之后，你会发现所有付出的时间和精力都是值得的。

1.4 JS Bin

JavaScript 通常在网页上运行。浏览器从服务器加载了一个可能包含 JavaScript 代码或超链接的网页，然后解析代码并执行指令。在《JavaScript 开发实战》的第一部分中，请读者避免过多考虑含有 JavaScript 代码的网页制作，而只需要专注 JavaScript 语言本身，完全可以利用免费在线服务 JS Bin。

JS Bin 是一个用于网页和 JavaScript 程序开发及共享的在线沙盒。本书中列出的所有代码清单都可以在 JS Bin (www.jsbin.com)中找到，JS Bin 为读者提供了编码练习和实践的机会。

初次访问该网址，会在标题部分看到 Dave 图片和一些帮助性的链接，这些链接可以帮助使用者熟悉和使用 JS Bin，如图 1-1 所示。大胆地去探索和学习吧！千万不要因为链接的信息过于复杂而放弃。浏览完毕，请点击 Dave 图片左侧的“X”按钮，关闭信息面板（使用者可以关闭 JS Bin 显示的任何欢迎信息或其他信息）。 5



图 1-1　JS Bin 上的各面板：HTML、CSS、Output 和信息面板

1.4.1 JS Bin 面板

JS Bin 是用于开发网页和应用程序的工具。除了位于顶部的信息面板，还有 5 个面板可供显示，分别是 HTML、CSS、JavaScript、Console（控制台）和 Output（输出）。单击 JS

Bin 工具栏上的面板名称可以打开或关闭该面板。在本书的第一部分，只需要使用 JavaScript 面板和 Console 面板；在第二部分，将使用 HTML 面板；在第三部分，将添加 CSS 和 Output 面板。本章只使用 JavaScript 和 Console 面板，因此先来学习如何切换这两个面板并关闭其他面板，如图 1-2 所示。



图 1-2　JS Bin 上显示的 JavaScript 面板和 Console 面板

1．HTML

6 HTML 用于构造网页的内容，如文字、图片、视频和表单都是网页的内容。

2．CSS

CSS（级联样式表）确定网页应该以什么样的外观向用户呈现，用于定义背景颜色、字体、边距和大小等。

3．JavaScript

JavaScript 向网页添加动作和交互，用于编写网页内容之外的程序。

4．Console

程序可以通过控制台向用户和开发人员显示信息，如错误提示和警告提示。控制台是交互式的，开发人员可以在控制台上输入信息来查找某个程序的状态。对于已经开发完成的应用程序，通常不再使用控制台，但在学习编程阶段，读者可以很好地利用它与程序进行简单便捷的交流。

5．Output

Output 面板可以预览已经在 HTML、CSS 和 JavaScript 面板中编好的网页。Output 面板显示的内容与网页访问者在浏览器中看到的内容完全一致。

1.4.2　在 JS Bin 上运行代码清单

开发人员通过向 JS Bin 的 JavaScript 面板添加代码来编写程序。随着代码功能的增加，程序将逐渐变得越来越复杂。对于本书第一部分中的大多数代码清单，可以按照以下步骤在
7 JS Bin 中进行测试，如图 1-3 所示。

图 1-3 是在 JS Bin 上运行 JavaScript 步骤的截图。本书中的大多数代码清单都有一个指向 JS Bin 上相同代码的链接。JS Bin 上的代码清单包括与之相关的额外信息和练习，将在 1.4.4 和 1.4.5 节进一步讨论。



图 1-3　在 JS Bin 上运行 JavaScript 的步骤

1.4.3　输出到控制台

如果希望程序能够在 Console 面板上输出信息，请使用 console.log 命令。

运行程序就会在 Console 上显示以下内容：

```
> Hello World!
```



代码清单 1-1　使用 console.log 显示信息 (http://JS Bin.com/mujepu/edit?js,console)

```
console.log("Hello World!");
```

如上所示，将欲显示的信息放在括号内的引号之间。

请注意，代码清单标题部分包含 JS Bin 链接，单击该链接可以查看在 JS Bin 上的代码。要在 JavaScript 面板中执行代码，请单击 Console 面板顶部的 Run 按钮，就会看到“Hello World！”出现在 Console 面板上。 8

读者可以尝试多次点击 Run。每次单击 Run，代码就会运行一次，“Hello World！”就会又一次出现在 Console 面板上。欲清除 Console 面板上的所有信息，请单击 Clear 按钮。

当通过 JS Bin 链接访问 JS Bin 上的代码时，程序会自动运行。如果你是 JS Bin 的注册用户，可以在 JS Bin 的首选项中关闭该自动运行选项。

1.4.4　代码注释

除了代码语句本身，本书的 JS Bin 代码清单还包括注释。注释文本并不是程序的一部分，但有助于读者理解代码。以下是 JS Bin 代码清单 1-1 的第一个多行注释：

```
/* Get Programming with JavaScript
* Listing 1-1
* Using console.log
*/
```

这些多行注释可以跨越多行。除了多行注释，有时也会看到单行注释，例如：

```
// This is a single-line comment
```

在 JS Bin 上，注释通常以绿色显示。如果程序开发者认为需要添加一些解释以供其他程序员理解，就可以在他们的代码中添加注释。当计算机执行程序时，会自动忽略这些注释。

1.4.5 Further Adventures——进阶练习

《JavaScript 开发实战》在 JS Bin 中的大多数代码清单都附带了一组名为 Further Adventures 的进阶练习，内容包括代码和注释。这些练习有些很简单，有些有重复，还有一些比较难，具有挑战性。学习编程的最好方法是实际动手去编程，所以希望读者能鼓足勇气，接受挑战。读者可以从 Manning Forums（曼宁论坛）获得帮助，并在以下网站得到本书所布置任务的解决方案：

```
www.manning.com/books/get-programming-with-javascript
www.room51.co.uk/books/getprogramming/index.html
```

1.4.6 错误消息

向 JavaScript 面板添加代码时，JS Bin 会不断检查代码中是否有错误，并将错误显示在 JavaScript 面板底部的一个红色区域内。开发者如果正处于添加一行代码的过程中，不必理会该红色错误区。如果已经完成了一行代码的添加，红色错误区依然存在，请单击该红色错误区，查看具体的错误消息。例如，尝试从代码清单 1-1 中的代码行末尾删除分号。图 1-4
9 显示了 JS Bin 响应删除分号时出现的错误。

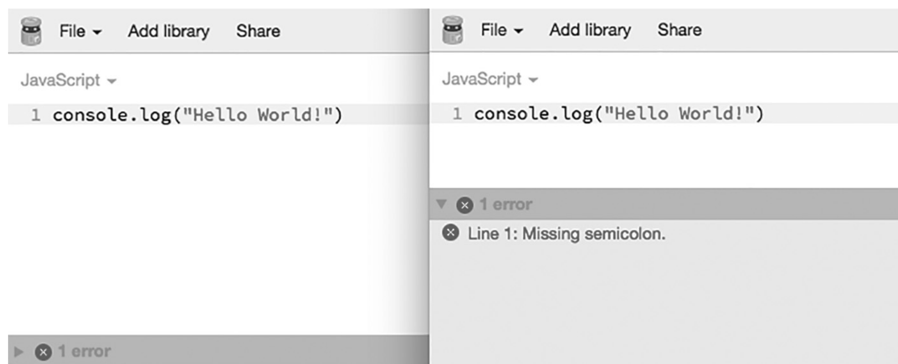


图 1-4 JS Bin 错误消息提示区域（收起状态和展开状态）

分号表示一行代码的结束。以分号结尾的每一行代码称为一条语句。如果停止输入代码，但没有用分号去结束，JS Bin 会报错。JavaScript 会在它认为应该有分号的地方插入分号，这样程序才会继续运行。但最好还是程序员自己把分号加上，在 JS Bin 中不断地对错误进行纠正可以提高程序开发者的编程实践水平。

JS Bin 会尽力提供错误消息以帮助使用者解决问题。如果从代码行的末端逐个删除字符，就会看到 JS Bin 不断更新报错信息。

1.4.7　行号

图 1-4 中的错误信息中提示了发生错误的行号。因为该示例中只有一行代码，所以错误就在第 1 行（Line 1）。当程序变得很长时，能够看到代码的行号就变得非常有意义了。程序开发者并不需要用手工方式为代码添加行号，文本编辑器 JS Bin 会为代码自动添加行号。这些行号并不是程序的必要组成部分，只是在编写和测试代码时能够为开发者提供帮助。如图 1-5 所示，在一个比较长的程序中存在两个错误。读者暂时不必理解该代码，只需要尝试着在 JS Bin 错误图表中找到这两个错误。如果程序变得更长又没有行号，程序员将很难找到错误的具体位置。

在 JavaScript 面板左上角有一个单词 JavaScript（见图 1-5），可以用来切换是否在 JS Bin 上显示行号。双击它可以打开或关闭一个菜单，同时也会使行号由出现切换到隐藏，或者由隐藏切换到出现。如果你是 JS Bin 的注册用户，可以在 JS Bin profile 中将行号设置为开启或关闭状态。



图 1-5　对查找错误有帮助的代码行号

1.4.8　获取账户

在 JS Bin 上注册一个免费的账户是很有价值的。注册账户后，你可以保存自己的程序，还可以有更多的个性化设置。当用户开始自己写程序时，就会发现 JS Bin 是一个可以大胆尝试新想法，并得到即时预览和反馈的绝妙之地。 10

1.5　游戏 The Crypt——本书的一个持续示例

本书从始至终一直在持续开发一个名为 The Crypt 的基于文本的冒险游戏。在这个游戏中，玩家将在地图上进行探索，从一个地方移动到另一个地方，得到能够帮助他们解决问题和跨越障碍的有用物品。在每章的最后一节，会使用刚学到的新内容对该游戏进行更新和完善。读者可以看到如何使用每章的编程知识来编写小程序，然后再将这些小程序组合生成一

个大程序。

1.5.1　运行 The Crypt

游戏将显示玩家的当前位置，以及在此地的所有物品（Items）和所有出口（Exits），如图 1-6 所示。

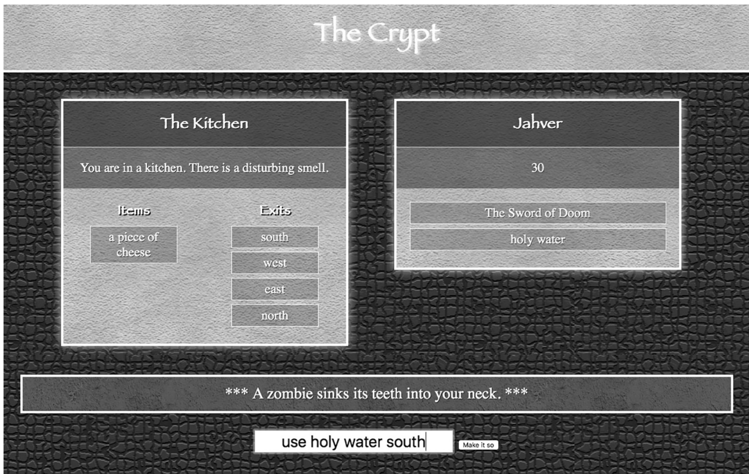


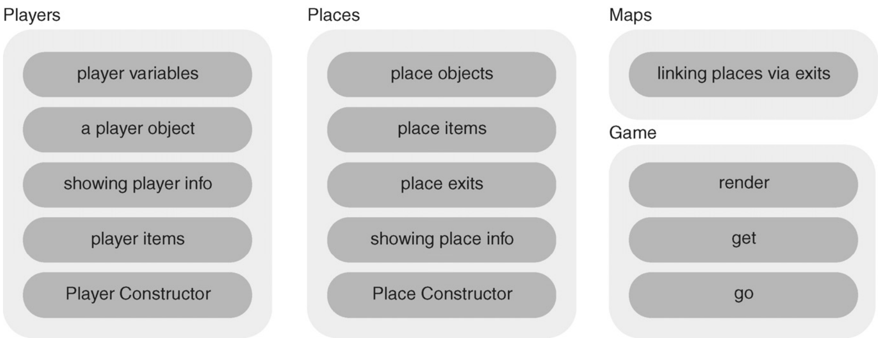
图 1-6　The Crypt 游戏界面

玩家可以输入命令，从一个地方移动到另一个地方，得到他们发现的物品，并使用这些物品来迎接挑战。

开发者需要为游戏中的所有元素编写代码。不过不要担心，本书会一步一步地教你如何
11 去做。请在 JS Bin 上运行该游戏：http://output.jsbin.com/yapiyic。

1.5.2　创建 The Crypt 的具体步骤

在本书的第一部分，我们已经学习了 JavaScript 的一些核心概念，现在可以编写代码为游戏中的玩家（Players）和地点（Places）命名，并可以让玩家在地点之间移动，以放置和拾取他们发现的物品。图 1-7 显示了该游戏需要创建的元素：玩家、地点、地图（Maps）和动作（Game）。请读者不必担心图中的术语，在本书的后续章节中会对其进行详细介绍。



12 图 1-7　The Crypt 在第一部分中的游戏元素

以下每一章都会有一个与上图类似的游戏元素图。第一部分和第二部分都将使用 JS Bin 上的控制台来显示游戏信息，并接受用户的输入。表 1-1 显示了游戏元素与第一部分内容的对应关系。

表 1-1　The Crypt 的游戏元素和第一部分中与之对应的 JavaScript

游 戏 元 素	任　务	JavaScript	章　节
玩家（Players）	确定我们所需要的每位玩家的信息	变量（Variables）	第 2 章
	收集在某一个地点中玩家的信息	对象（Objects）	第 3 章
	在控制台上显示玩家的信息	函数（Functions）	第 4～7 章
	创建每个玩家收集到的物品列表	数组（Arrays）	第 8 章
	组织创建玩家的代码	构造函数（Constructors）	第 9 章
地点（Places）	创建结构类似的探索地点	构造函数（Constructors）	第 9 章
	连接地点与出口	方括号运算符（Square bracket notation）	第 10 章
动作（Game）	添加简单的功能：移动、收集物品和显示信息	方括号运算符（Square bracket notation）	第 10 章
地图（Maps）	连接地点与出口	方括号运算符（Square bracket notation）	第 10 章

本书第二部分为玩家增加了挑战，在玩家使用合适的物品解出谜题之前，无法打开出口。第二部分编程的重点集中在以下几个方面：代码的组织、程序的部分隐藏、用户输入的检查以及能够增加项目灵活性的可重复使用和替换的模块。

图 1-8 显示如何将该游戏拆分为地图数据和用于创建玩家和地点的构造函数模块、用于在控制台上显示信息的视图以及用于运行游戏并链接所有程序块的控制器。再次指出，目前读者不需理解图表中的所有术语，只要保持好奇和兴奋就好！所有上述概念将在本书的后续章节中详细叙述。 13

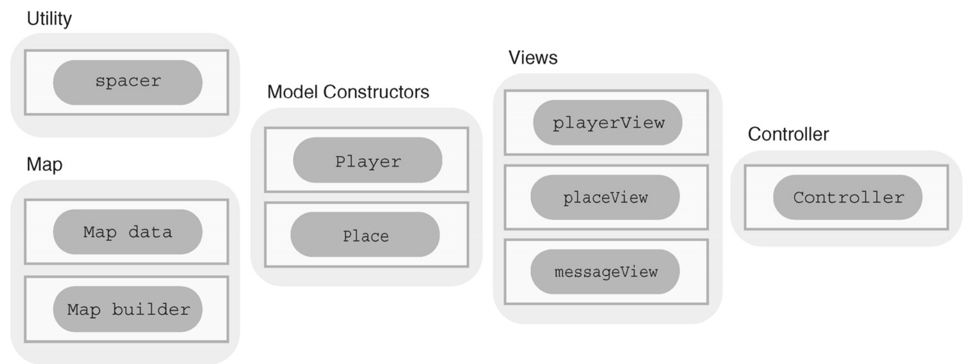


图 1-8　第二部分中的 The Crypt 游戏元素

本书第三部分讲解如何使用 HTML 模板更新该游戏，如何在游戏运行中加载数据，如何使用玩家和地点信息填充模板，以及如何使用文本框和按钮让玩家能通过网页输入命令，如图 1-9 所示。

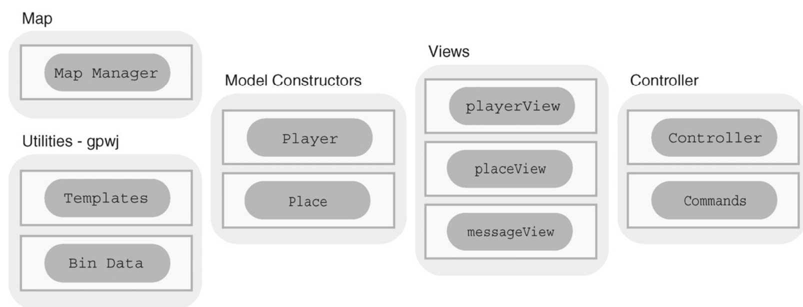


图 1-9　第三部分中的 The Crypt 游戏元素

1.6　更多示例和练习

尽管游戏 The Crypt 是我们持续学习 JavaScript 的程序背景，但是在本书的每一章也包括了许多其他示例，帮助读者掌握所学基本概念在各种情况下的运用。伴随着学习的一步步
14 深入，也会逐渐开发一些其他程序示例，以帮助读者理解新概念如何优化和改进程序。在后续章节中，我们会看到一个测验应用程序、一个健身记录应用程序、一个电影评分网页和一个新闻标题网页。

1.7　浏览器的兼容性

浏览器一直处在不断升级中，低版本的浏览器（如 Internet Explorer 8 和其他更老的浏览器）可能不支持本书中的某些 JavaScript 代码。在本书的在线论坛中，可以找到上述问题的解决方案。

1.8　本章小结

- 程序是一组计算机遵循的指令集合。
- 高级语言让我们写出更容易阅读和理解的指令。
- 作为世界上最广泛使用的编程语言之一，JavaScript 为网页增加了很强的交互性。作为应用程序的脚本语言，JavaScript 也可用于服务器端编程、机器人编程和其他设备编程。
- 任何学习过程都遵循思维规律，循序渐进才能取得进步。请读者认真练习本书的示例，并保持好奇心，深入挖掘，百折不挠。
- JS Bin 是一个在线代码沙盒，将帮助读者聚焦于 JavaScript 语言，在进行代码实验和练习过程中为其提供快速反馈。
- 本书的主要持续示例是 The Crypt。该游戏为读者学习编程概念提供了程序环境，并且示范了如何从简单元素起步，逐渐编写出一个相对复杂的程序。
- 其他更多示例将拓展读者编程的广度和深度，并有助于读者看到自己所学到的编程
15 知识可以应用到更广泛的领域。

第 2 章　变量：在程序中存储数据

本章内容包括：

- 用变量存储和使用信息
- 声明变量
- 为变量分配值
- 在控制台上显示变量值

《JavaScript 开发实战》是一本介绍编程的入门书，本章可以看作是这本入门书的起点。千里之行始于足下，冒险之旅不能弃家不谈，因为家是你收拾行囊、开启冒险之旅的起点，它虽然不是冒险的主要内容，但至关重要，就好像去机场不能忘记带护照，去参加奥斯卡颁奖典礼不能忘记拿自拍杆一样。

无一例外，程序存储、操作和显示的对象是数据。无论是编写博客系统、分析发动机性能、预测天气还是设计 10 年以后要发送到彗星上着陆的探测器，都需要考虑所需使用的数据以及数据可能采用的类型。要在程序中使用数据，就必须使用变量。 16

2.1　什么是变量

在程序中，变量（variable）就是被命名的值。当程序员在程序中使用该名称时，会被替换为值。例如，程序中创建了一个名为 score 的变量，并对其赋值 100。之后，如果给计算机发送指令“显示 score”，计算机就会显示 100。因为变量是可以变化的，所以在后面的程序中，score 变量可以对玩家的某些动作做出响应，并随时对 score 变量的值进行更新。例如，先将 score 变量的值增加 50，然后让计算机“显示 score”，计算机就会显示 150。

如何使用 JavaScript 来实现上述小戏法？

2.2　变量的声明和赋值

为了让计算机理解所存储的信息，通常需要以下两个步骤：

首先，设置一个变量名称，并使用这个名称来引用程序中的数据，例如 score、playerName 或 taxRate。

然后，在变量名称与它所存储的值之间建立起连接，例如设置 score 等于 100、设置玩家名字是 George 或者将税率设置为 12%。

在 2.2.3 节中，我们将学习如何使用一条 JavaScript 语句完成以上两个步骤。 但是现在，我们先来学习如何使用两条语句完成以上两个步骤。

2.2.1　变量声明

在游戏中，我们常常需要记录不断变化的得分，完全可以用程序来追踪这些变化的分数，这就意味着需要设置一个变量。

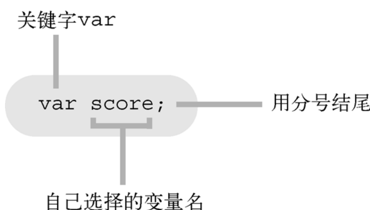
变量声明就是用一个名称来表示值。在 JavaScript 中，使用关键字 var 来声明变量。以下代码中声明了一个名为 score 的变量：



代码清单 2-1　声明变量
(http://jsbin.com/potazo/edit?js,console)

```
var score;
```

关键字 var 命令计算机从该语句中取出下一个单词，并将其变成一个变量。图 2-1 是对代码清单 2-1 各部分的详解。



17

图 2-1　声明变量

图 2-1 显示了变量声明语句中各部分的名称。就这样，我们成功地声明了一个名为 score 的变量。

2.2.2　变量赋值

现在，程序已经能够识别变量 score，下面介绍如何为变量 score 分配一个值。为变量赋值需要使用等号。图 2-2 说明了等号的作用，代码清单 2-2 显示了等号在程序中的作用。

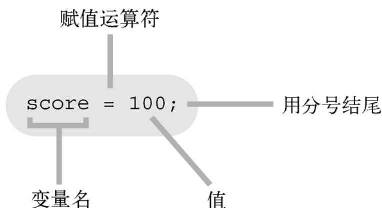


图 2-2　为变量赋值



代码清单 2-2　为变量赋值
(http://jsbin.com/yuvoju/edit?js,console)

```
var score;          //声明一个名为 score 的变量
score = 100;        //将 100 赋值给 score
```

通过以上两条语句，我们将 100 赋值给变量 score。在赋值语句中，将值放在等号的右边，将变量放在等号的左边（图 2-3）。当使用等号为变量赋值时，将等号称作赋值运

算符。



图 2-3　等号称作赋值运算符

现在，我们已经学会了声明一个变量并对其进行赋值。下面介绍如何将变量的值显示在控制台上。以上代码清单 2-2 的输出如下所示：

```
> 100
```



代码清单 2-3　使用变量
(http://jsbin.com/huvime/edit?js,console)

```
var score;
score = 100;
console.log(score);
```

18

利用第 1 章中介绍的 console.log 函数，可以在控制台上显示变量 score 的值。因为我们已经将 100 赋值给了 score，所以在控制台上就会显示 100。

为什么不直接使用 console.log(100)来显示 100？因为在程序运行中变量的值通常是不断变化的。通过使用变量，而不是字面值，程序可以随时取到这个变量中存放的值。以下代码清单 2-4 首先在控制台上显示 score 变量中存放的值，然后更改 score 变量的值，并显示更改后的新值，输出结果如下：

```
> 100
> 150
```



代码清单 2-4　变量的值发生变化
(http://jsbin.com/jasafa/edit?js,console)

```
var score;
score = 100;
console.log(score);
score = 150;
console.log(score);
```

在以上代码中，两次使用相同的指令：console.log(score)，但是在控制台上输出了两个不同的值。尽管代码指令中使用了相同的变量 score，但因为变量 score 的值前后发生了变化，因此输出也会相应改变。

在以上代码中，程序分两次将 100 和 150 赋值给变量 score。与之类似，程序指令也可以轻松地将文本内容赋值给变量，只需将文本内容放在引号中即可。以下代码清单 2-5 可以

在控制台上显示以下两条信息：

```
> Hello World!
> Congratulations! Your tweet has won a prize ...
```



代码清单 2-5　将文本赋值给变量
(http://jsbin.com/hobiqo/edit?js,console)

```
var message;                    //声明变量 message
message = "Hello World!";       //使用双引号将一条文本信息赋值给变量 message
console.log(message);
message = 'Congratulations! Your tweet has won a prize...';
                                //使用单引号将一条新的文本信息赋值给变量 message
console.log(message);
```

正如代码清单 2-5 所示，将文本放在引号内就称为一个字符串。我们可以使用双引号，例如"Hello World!"；或者使用单引号，例如'Congratulations!'。但是单引号或双引号必须成对
19 出现。如果没有使用引号，JavaScript 会将文本信息解释为指令或变量。

2.2.3　一步实现变量声明和赋值

在以上两节中，读者已经学会先声明变量，然后再为变量赋值。下面，讲述如何在一条语句中既声明一个变量又为其赋值，如图 2-4 所示。



图 2-4　在一条语句中声明一个变量并为其赋值

代码清单 2-6 和 2-7 实现了以下完全相同的结果：声明变量、为变量赋值，然后显示以下消息：

```
> Kandra is in The Dungeon of Doom
```



代码清单 2-6　分两步实现先声明变量，然后再为变量赋值
(http://jsbin.com/vegoja/edit?js,console)

```
var playerName;
var locationName;
playerName = "Kandra";
locationName = "The Dungeon of Doom";
console.log(playerName + " is in " + locationName);    //使用加号来连接字符串
```



代码清单 2-7　一步实现变量声明和赋值
(http://jsbin.com/dorane/edit?js,console)

```
var playerName = "Kandra";    //一步实现变量声明和赋值
```

```
var locationName = "The Dungeon of Doom";    //一步实现变量声明和赋值
console.log(playerName + " is in " + locationName);
```

在代码清单 2-7 中，程序将每个等号右侧的值赋给等号左侧那个新声明的变量。在这两个程序中，通过使用加号连接多段文本并把它们显示在控制台上。将多段文本连接在一起称作“字符串连接”，加号称作“字符串连接运算符”。

如果在声明变量时就已经知道它的值，就可以采用一步实现变量声明和赋值的便捷方式。有时，在声明变量时并不知道它的值，可能需要经过计算、需要用户输入或者需要等待网络响应。在这些情况下，声明变量和赋值就要分开进行。通常，程序员习惯在程序开始处就声明变量，即使他们后来并不会为其赋值。 20

2.2.4　先运算再赋值

在为变量赋值时，JavaScript 先计算赋值运算符右边的表达式，再将计算结果赋值给该变量。

```
var score;
score = 100 + 50;
```

JavaScript 先计算表达式：100 + 50，然后将结果 150 赋值给变量 score。

表达式中的值有可能不是诸如 100 和 50 这样的硬编码字符串，而是变量。在下面的例子中，使用变量 callOutCharge，costPerHour 和 numberOfHours 来计算雇用水管工的总成本：

```
total = callOutCharge + costPerHour * numberOfHours;
```

`*` 符号用于乘法，称作乘法运算符。 当然，我们也可以使用 `-` （用于减法）和 `/` （用于除法）。

因为 JavaScript 首先计算赋值运算符右边的表达式的值，然后再将计算结果赋值给左边的变量，因此可以使用变量的当前值来设置其新的值。例如在游戏应用程序 The Fruitinator（切水果）中，玩家每切中一个草莓就可以得到 50 分！玩家需要不断地更新得分。

```
> Your score was 100
> Great splat!!!
> New score: 150
> Way to go!
```

图 2-5 显示了一条使用变量的当前值来设置其更新的值的语句，代码清单 2-8 是一个对得分进行更新的程序。

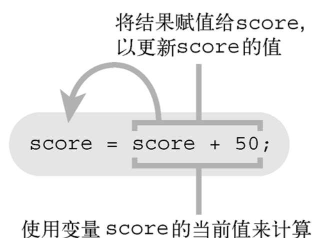


图 2-5　用包含变量本身的计算结果来更新变量的值



代码清单 2-8　使用变量的当前值来设置其更新值
(http://jsbin.com/kijuce/edit?js,console)

21

```
var score;
score = 100
console.log("Your score was " + score);
console.log("Great splat!!!");
score = score + 50;    //将当前 score 加 50，然后将计算结果赋值给变量
console.log("New score: " + score);
console.log("Way to go!");
```

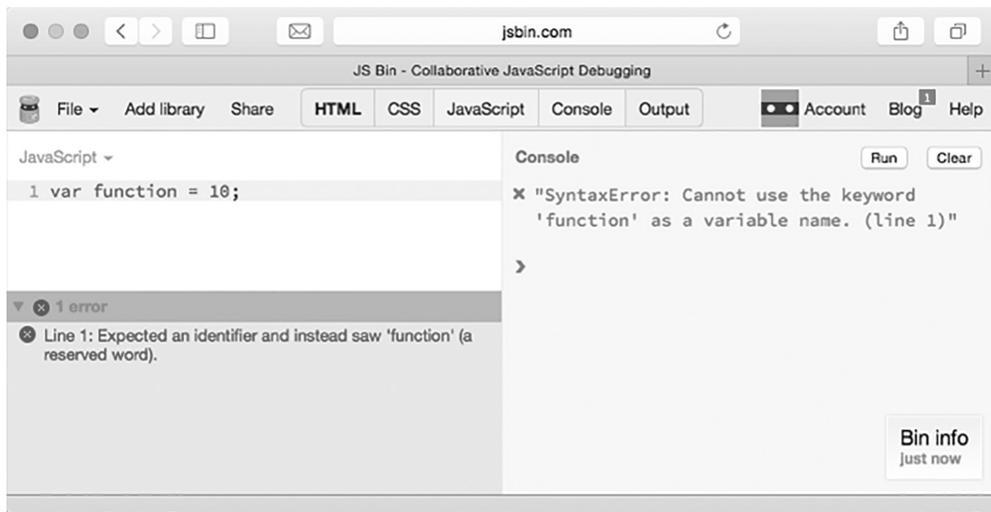
在以上代码中，首先使用 score 的当前值 100，用表达式 score + 50 计算出结果是 150，然后将 150 赋值给变量 score。

2.3　选择合适的变量名

在上述所有的代码清单中，并无意强迫编程者必须使用上述这些变量名，但是确实应该尽量选择一些让读者容易理解的变量名。编程者基本上可以自由选择变量名，但是必须避免 JavaScript 内部已经使用的关键字和保留字。

2.3.1　关键字和保留字

JavaScript 有一些关键字，例如 var 和 function。关键字是 JavaScript 语言本身的一部分，管理每个程序中的操作和属性。JavaScript 也设置了一些保留字，它们将来可能会成为 JavaScript 语言的关键字。我们不能使用这些关键字或保留字作为变量的名称。其他关键字有 if、switch、do 和 yield 等，Mozilla Developer Network 上公布了完整清单供学习者查询（http://mng.bz/28d9）。读者可以不接受我的忠告，到 JS Bin 上尝试使用清单中的关键字作为变量名称，那么就会看到如图 2-6 所示的信息。



22

图 2-6　在 JavaScript 中有一些词不能用作变量名

读者不必完全知晓关键字和保留字的清单。随着编程经历的增长，你自然会知道哪些是关键字，而且当你尝试使用它们作变量名时，通常会有错误提示。如果你的程序无法运行，你又找不到原因时，就有必要考虑这个因素。

2.3.2　变量的命名规则

在上一节中，我们知晓不能采用关键字和保留字作为变量名，是否意味着其他词都可以用作变量名？不完全是。关于变量的命名还有一些其他规则。 变量名的第一个字符可以是任何字母、美元符号“$”或下画线“_”。变量名的后续字符可以是以上任何一个字符或数字。但是不允许空格。代码清单 2-9 中包括一个有效变量名和一个无效变量名。如果访问 JS Bin 上的代码，将会看到一长串错误报告。请读者尝试阅读并理解清楚这些错误报告。如果确实无法理解也不必担心，因为该代码清单中包括了许多无效的变量名，JavaScript 对此很不高兴。



代码清单 2-9　有效的变量名和无效的变量名
(http://jsbin.com/biqawu/edit?js,console)

```
var thisIsFine;
var $noProblemHere;
var _underscore56;
var StartWithCapital;
var z5;        //以上是有效的变量名
var 999;       //以下是无效的变量名
var 39Steps;
var &nope;
var single words only;
var yield;
```

JavaScript 区分大小写字母。更改变量名中的大小写字母就意味着一个新的变量，例如 score、Score 和 SCORE 是三个不同的变量名。以上这些差异有时候很难发现，所以需要尽量保持一致，详见以下 2.3.3 节的内容。

2.3.3　骆驼式命名法（camelCase）

仔细观察 camelCase 中大写字母的使用位置。与之类似，很多变量名（例如 costPerHour、playerName 和 selfieStickActivated）都是由多个词连接在一起构成的，其中第一个词是小写字母，后续的词以大写字母开头，这种命名方式称为骆驼式命名，又称驼峰命名法。这种命名法更易于程序员在同行之间交流代码，因而被广泛使用。

有些程序员愿意使用下画线来分隔变量名中的各个单词，例如 cost_per_hour、player_name 和 selfie_stick_activated。其实，如何为变量命名完全取决于程序员自己，属于个人编程的风格。本书采用骆驼命名法。

23

2.3.4　使用描述性变量名

我们可以使用描述用途或描述目的的词为变量命名。虽然编程者确实可以自由选择变量名，但是很显然 costPerHour 比 cph 更易于理解。其他程序员将来有可能需要读取和更新你

的代码，你自己将来也可能会再读自己的代码，那时候，你就会感谢你自己曾经使用了这些容易理解的变量名。随着程序越来越长，就会涉及越来越多的变量、对象和函数，合适的变量名有助于开发者追踪程序的流程并理解其目的。所以，为变量命名时要尽量做到简单、直接和易读。

现在，读者已经理解了变量的概念，并学会了如何声明和赋值变量，还明白了如何为变量选取一个合适的名称。下一步，读者需要了解如何在程序中确定变量。首先，需要分析问题；然后，规划一个适合用户的解决方案。在下一节中，我们需要考虑游戏 The Crypt 中代表玩家的各种信息。

2.4 The Crypt——玩家变量

在第 1 章中，我们已经知道游戏 The Crypt 中包括许多元素：玩家（Players）、地点（Places）、动作（Games）、地图（Maps）和挑战（Challenges）。在设计和构建游戏时，就需要考虑所有这些元素的属性。如图 2-7 所示，突出显示了玩家变量。

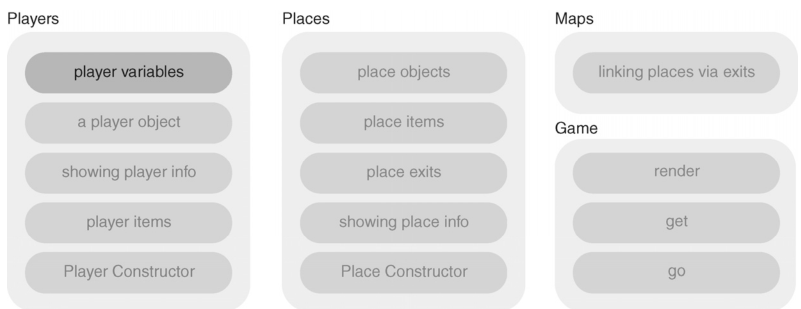


图 2-7　游戏 The Crypt 中包括的元素

随着玩家从一个地点移动到另一个地点，程序需要知道创建些什么内容才能使冒险既有趣又富挑战性。我们可能想要跟踪姓名、健康、携带的物品或位置，或者也可能想要跟踪一下玩家脚上的毛和激光剑的颜色。这些信息中有一些内容可能在游戏中一直保持不变，但是
24 有一些可能会变化。

编程技艺之美来源于懂得取舍，知道程序应该包括哪些信息，又应该剔除哪些信息。例如玩家的脚上到底有多少根毛也许会对游戏产生一些影响，但是我们对此无法知晓。游戏的开发者应该仔细考虑玩家在完成任务时，应该使用什么数据。

表 2-1 显示了每一个玩家身上可能会拥有的属性。

表 2-1　玩家可能会拥有的属性

属　性	描　述	示 范 值
名字	用于显示玩家的信息，用于与其他玩家交互时使用	"Kandra", "Dax"
健康	因怪物和毒物而减少，因食物和药品而增加	68
地点	玩家在地图上的哪个地点？	"The Old Library"
脚毛	衡量玩家赤脚在寒冷环境中应对水平的高低	94
物品	跟踪玩家已经得到的物品	"A rusty key"，"A purple potion"，"Cheese"

游戏可能还需要其他属性，那么就根据需要添加和删除属性即可。以下是对玩家属性的声明：

```
var playerName = "Kandra";
var playerHealth = 50;
```

程序员的专业素养体现在能够针对具体情况建立程序模型，并预测需要哪些变量来完成一个程序。预测得越好，将来大面积重写程序的概率就会越小——没有人愿意这么做，正如没有人愿意到了飞机场才发现自己忘记带护照了，同样没有人愿意在已经写了很多代码之后才发现自己忽略了一个重要部分。

2.5　本章小结

- 变量可以让程序在运行中存储数据。
- 通过在关键字 var 之后加一个名称的方法来声明变量：

```
var costPerHour;
```

- 为变量选择简单并且合适的名称，避免使用 JavaScript 的关键字和保留字。 25
- 使用赋值运算符（=）为变量赋值：

```
costPerHour = 40;
```

该语句将等号右边的值赋给等号左边的变量。

- 在表达式中使用变量：

```
total = callOutCharge + costPerHour * numberOfHours;
```

- 作为程序规划设计的一部分，需要认真思考声明什么变量，以及这些变量将拥有什么类型的数据。 26

第 3 章　对象：数据分组

本章内容包括：

- 使用 JavaScript 对象组织信息
- 创建对象
- 为对象添加属性
- 使用圆点运算符访问属性
- 对象的示例

在第 2 章中，我们学习了如何声明变量，以及如何为变量赋值。在游戏 The Crypt 中，使用变量为玩家建模。随着程序越来越长，使用变量的数量自然也会越来越多，因此需要一些方法来组织这些数据，使程序更加容易理解、容易更新并且将来更加易于移植到其他程序中去。

在生活中，我们常常对诸多物品进行分组，然后把每组物品作为一个整体来看待。以急救包为例，急救包内装了很多物品，但是，医护人员通常都会把急救包作为一个整体来对待。他们会说“你带好急救包了吗？”“把急救包给我。”“我们需要一个急救包！快！”只有当他们真正需要急救包内的物品时，才会把关注的焦点切换到急救包里面的内容，例如“把包里的消毒剂和绷带递给我。”在上述例子中，人们使用急救包这个单个对象封装了许多零碎物品。

本章将介绍 JavaScript 语言中的对象。对象是一种简单、有效的收集变量的方法，在程序中可以把这些变量看作一个组，而不是各个单独的变量来进行信息传递。

3.1　变量需要分组

你陆陆续续买了很多冒险故事书，现在，你决定编写一个程序来管理这些藏书。以下代码清单 3-1 将一本书的信息显示在控制台上，如下所示：

```
> The Hobbit by J. R. R. Tolkien （《霍比特人》作者：J. R. R. Tolkien）
```



代码清单 3-1　使用变量表示一本书
(http://jsbin.com/fucuxah/edit?js,console)

```
var bookTitle;      //声明变量
var bookAuthor;
bookTitle = "The Hobbit";     //对变量赋值
bookAuthor = "J. R. R. Tolkien";
console.log(bookTitle + " by " + bookAuthor);//用变量来显示此书的信息
```

在以上代码中，首先使用关键字 var 来声明两个变量：bookTitle 和 bookAuthor，后续程序中将使用这两个变量名来存储和访问值；然后将字符串（文本）赋值给这两个变量，引号内的字符串不会被 JavaScript 解释为关键字或变量名；最后，使用加号将三个字符串连接在一起并将其显示在控制台上。

在上例中，使用两个变量显示了一本书的信息。随着你购买的书籍越来越多，又该如何为这些书籍声明变量？答案是为每个变量名加上不同的前缀。以下代码清单将三本书的信息显示在控制台上：

```
> There are three books so far ... （目前有三本书）
> The Hobbit by J. R. R. Tolkien（《霍比特人》作者：J. R. R. Tolkien）
> Northern Lights by Philip Pullman（《北极光》作者：Philip Pullman）
> The Adventures of Tom Sawyer by Mark Twain （《汤姆·索亚历险记》作者：
Mark Twain）
```



代码清单 3-2　使用前缀来区分变量名

```
var book1Title = "The Hobbit";     //一步实现变量声明和变量赋值
var book1Author = "J. R. R. Tolkien";
var book2Title = "Northern Lights";
var book2Author = "Philip Pullman";
var book3Title = "The Adventures of Tom Sawyer";
var book3Author = "Mark Twain";
console.log("There are three books so far...");
console.log(book1Title + " by " + book1Author);
console.log(book2Title + " by " + book2Author);
console.log(book3Title + " by " + book3Author);
```

28

以上方法成功解决了三本书的问题，应该说这种方法在一定数量范围内奏效，但是随着书目的数量继续增加，以及每一本书所包含信息量的继续增加，变量的数目会大得惊人，甚至会失控！因此需要使用单个变量将一本书的所有信息集合起来，这就是下一节的内容——对象。

3.2　创建对象

对象 book1 就像急救包一样，是一个打包件，而各个变量 book1Title、book1Author、book1ISBN，就如同急救包内的剪刀、消毒剂、绷带、膏药等零散件。显而易见，打包件比零散件易于处理。因此，JavaScript 为我们提供了创建对象的能力，对象即变量组。下面，我们看一个完整的例子，然后再把它分解成各个不同部分。

代码清单 3-3 显示如何将一本书创建为一个对象，而非一堆零散变量的集合。图 3-1 显示了对象 book 在 JS Bin 控制台上的输出形式。



代码清单 3-3　对象 book
(http://jsbin.com/funiyu/edit?js,console)

```
var  book;
```

```
book = {
    title : "The Hobbit",
    author : "J. R. R. Tolkien",
    published : 1937
};
console.log(book);
```



29 图 3-1　在 JS Bin 控制台上输出一个对象

在 JS Bin 上运行代码清单 3-3，控制台在显示对象的同时，还会显示该对象的所有属性。请注意，控制台会按照字母顺序显示属性。对象本身并没有为各个属性排序，只是 JS Bin 在显示时对各个属性进行了排序。

下面将详细介绍如何创建对象以及各种运算符号的含义。

3.2.1　创建一个空对象

在第 2 章中我们看到，可以先创建一个变量，在后续程序中再对该变量进行赋值。此赋值可能来源于用户输入、服务器的响应、传感器的读取。同理，也可以先创建一个没有属性的对象，在未来需要的时候再为其添加属性。

用花括号创建一个对象，如下所示：



代码清单 3-4　创建一个空对象
(http://jsbin.com/funiyu/edit?js,console)

```
var book;              //使用关键字 var 声明一个变量
book = {};             //用花括号创建一个对象，并将其赋值给该变量
```

上述代码创建了一个没有任何属性的空对象，并将其赋值给变量 book。没有任何属性的对象是毫无用处的，在 3.4 节中将介绍如何为一个已经存在的对象添加新属性。下面介绍如何创建一个具有属性的对象。

3.2.2　属性“键–值对”

在代码清单 3-3 中，书籍对象包含三个属性：书籍标题、书籍作者、书籍的出版年份。这些属性的值分别是“The Hobbit”、“J. R. R. Tolkien”和“1937”。在 JavaScript 对象中，我们把属性的名称叫作“键”。book 对象中的键分别是 title、author、published。在为对象添加

属性时，使用花括号将键和值括起来，键与值之间加入冒号，如图 3-2 所示。

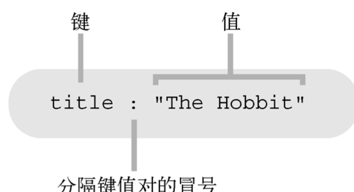


图 3-2　用键-值对来设置属性

30

键-值对有时也称作“名称-值对”，但本书中一直使用“键-值对”。

在下列代码清单中，创建了一个包含单个属性的对象。



代码清单 3-5　一个单个属性的对象
(http://jsbin.com/funiyu/edit?js,console)

```
var book;
book = {
  title : "The Hobbit"
};
```

在上述代码中，先声明一个变量，再创建一个对象，并将这个对象赋值给该变量。该对象仅有一个属性，该属性的键是“title”，该属性的值是“The Hobbit”。简单地说，对象 book 的属性 title 值为“The Hobbit”。

属性的值并不仅限于数字和字符串，例如“50”或者“The Hobbit”。还可以使用以前声明过的变量作为属性的值。以下代码清单把一本书的名称赋值给一个变量，然后使用该变量作为对象的属性值。



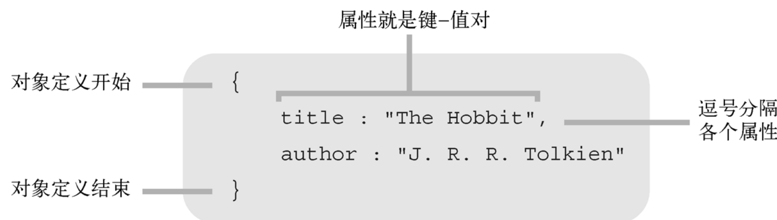
代码清单 3-6　变量作为属性的值
(http://jsbin.com/funiyu/edit?js,console)

```
var book;
var bookName;
bookName = "The Adventures of Tom Sawyer";   //把书名赋值给变量
book = {
   title : bookName   //该变量作为属性的值
};
```

将上述把变量作为对象属性的方法应用在对象只有一个属性的情况，貌似有些奢侈，与其这样复杂，还不如简简单单，直接使用一个变量。下面介绍如何创建一个拥有多个属性的对象。

当对象有多个属性时，请使用逗号来分隔键-值对。图 3-3 显示了拥有两个属性的对象。代码清单 3-7 创建了两个对象，每个对象拥有两个属性。

```
{
   title : "The Hobbit",
   author : "J. R. R. Tolkien",
};
```



31 图 3-3 创建一个拥有两个属性的对象



代码清单 3-7 拥有多个属性的对象
(http://jsbin.com/funiyu/edit?js,console)

```
var book1;
var book2;
book1 = {
   title : "The Hobbit",                //用逗号分隔各个属性
   author : "J. R. R. Tolkien"
};
book2 = {
   title : "Northern Lights",           //用键-值对设置属性
   author : "Philip Pullman"
};
```

现在我们已经学会如何创建对象，下面学习如何访问对象的属性。

3.3 访问对象的属性

对象的概念与急救包非常相似。我们可以把急救包从一个人传递给另一个人，从一个地方拿到另一个地方。只有当我们需要使用这个急救包内的物品时，才知道急救包里面有消毒剂、剪刀、绷带等物品。

对于 JavaScript 对象，可以使用圆点运算符来访问对象的属性值，具体形式是使用圆点运算符将存放对象的变量名称和对象的属性名称（键）连接在一起。如果把急救包作为一个对象（kit 是存放该对象的变量），可以使用 kit.antiseptic、kit.scissor 或 kit.bandages 来访问属性值。把书作为一个对象，如果要访问 book（存放该对象的变量）的 author 属性值，可以写成 book.author，如图 3-4 所示。



图 3-4 使用圆点运算符访问对象的属性

以下代码清单 3-8 将 book 对象的 title 和 author 属性显示在控制台上，如下所示：

```
> The Hobbit
> J. R. R. Tolkien
```



代码清单 3-8　使用圆点运算访问对象的属性值
(http://jsbin.com/funiyu/edit?js,console)

```
var book;
book = {
    title      :  "The Hobbit",    //使用键-值对为对象属性赋值
    author     :  "J. R. R. Tolkien",
    published  :  1937
};
console.log(book.title);   //使用圆点运算符访问属性值
console.log(book.author);
```

32

在上述代码清单 3-8 中，使对象的属性能够整齐排列的冒号可以增加程序的可读性。JavaScript 运行时会自动忽略那些多余的空格。整齐的代码块和排成一行的属性值可以使程序更容易阅读和理解，特别是当程序代码大规模增长的时候，这种整洁的形式使代码更加易于阅读、易于维护并易于更新。

显然，使用一个对象来替换一堆单个变量可以使程序更加简洁。这种做法使我们能够清晰地看到程序如何运行，而那些藏在内部的细节只有在需要时才去查看。在程序中，书籍被当作一个整体看待，只有在需要的时候，才会去访问书籍名称、书籍作者或出版日期。在本章中，用一个拥有 3 个属性的变量来替换 3 个单独变量，这似乎也算不上是很大的改进，但是当我们开始使用第 7 章的函数对象和第 8 章的数组对象时，就会看到使用对象所带来的简洁性和明晰度。

我们可以像使用变量一样来使用属性值。下列代码清单 3-9 将输出书籍名称和书籍作者，二者之间由字符串“by”相连：

```
> The Hobbit by J. R. R. Tolkien
> Northern Lights by Philip Pullman
```



代码清单 3-9　连接多个字符串属性
http://jsbin.com/yoweti/edit?js,console

```
var book1;
var book2;
book1 = {
    title: "The Hobbit",
    author: "J. R. R. Tolkien"
};
book2 = {
    title: "Northern Lights",
    author: "Philip Pullman"
};
console.log(book1.title + " by " + book1.author);
console.log(book2.title + " by " + book2.author);
```

3.4 更新对象的属性

在一个小测验的应用程序中，玩家针对一个个问题进行回答。问题的总数、玩家答对问题的数量、以及玩家的得分都会随时间的变化而变化。我们可以创建一个玩家对象，使其具有一系列初始值，然后每当他回答一个新问题，便对这一系列值进行更新。使用圆点运算符
33 可以更改已经存在的属性，或者为对象添加新的属性。如以下代码所示：



代码清单 3-10　使用圆点运算符更新对象的属性
(http://jsbin.com/mulimi/edit?js,console)

```
var player1;
player1 = {
    name: "Max",           //在创建对象时设置初始属性
    attempted: 0,
    correct: 0,
};
player1.attempted = 1;  //使用圆点运算符来更新属性
player1.correct = 1;
player1.score = 50;     //添加一个新属性并对其赋值
```

在以上代码中，在创建对象时，为 attempted 和 correct 属性设置了初始值 0，后来将这些属性的值更新为 1。程序使用了赋值运算符 =。等号将其右边的值 1 赋值给等号左边的属性 player1.attempted。 代码中设置了 attempted 和 correct 两个属性，然后立即对它们进行了更新。实际上，玩家每回答一个问题，程序就会做一次更新响应。

我们也可以在创建对象后再向其添加新属性。在上述代码清单 3-10 中，将 50 赋值给对象 player1 的新属性 score：

```
player1.score = 50;
```

在最初创建对象 player1 时，并没有设置属性 score。为一个本来不存在的属性 score 赋值意味着自动创建了该属性。

我们可以在计算中使用一个变量，并将计算结果再赋值给该变量。以下代码清单 3-11 显示了如何更新玩家的属性：

```
> Max has scored 0
> Max has scored 50
```



代码清单 3-11　先计算再赋值
(http://jsbin.com/cuboko/edit?js,console)

```
var player1;
player1 = {
    name: "Max",
    score: 0
};
```

```
console.log(player1.name + " has scored " + player1.score);
player1.score = player1.score+50;//先计算等号右边的表达式，再将计算结果赋值给该属性
console.log(player1.name + " has scored " + player1.score);
```

34

更新属性 score（在代码中以粗体显示）时，JavaScript 会先计算等号右边的表达式。因为 player1.score 为 0，所以表达式变为 0+50，得到 50；然后 JavaScript 将 50 赋值给等号左边，也就是属性 score。就这样，将 player1.score 从 0 更新为 50。

3.5　其他示例

尽管开发 The Crypt 程序能够给读者一个持续的机会来学习和练习 JavaScript 新概念，但是其他更多的示例将有助于读者拓展学习 JavaScript 的深度和广度。以下示例中都使用花括号来创建对象，然后仅仅使用一个步骤将该对象赋值给一个使用 var 关键字创建的变量。

3.5.1　撰写一条博客

每一条博客都是由若干条博客帖子组成。每一条帖子都包括作者信息、创建日期、评论等内容。下面是用一个对象表示一条帖子：



代码清单 3-12　一条博客帖子
(http://jsbin.com/jiculu/edit?js,console)

```
var post = {
   id : 1,
   title : "My Crazy Space Adventure",
   author : "Philae",
   created : "2015-06-21",
   body : "You will not believe where I just woke up!! Only on a comet..."
};
```

3.5.2　创建一个日历

日历事件一定会包含若干个约会。JavaScript 确实有一个现成的 Date 对象，用于处理日期和时间，但是在本书中并不会使用它。以下代码中用特定格式的字符串来表示约会。



代码清单 3-13　一个日历事件
(http://jsbin.com/viroho/edit?js,console)

```
var event = {
   title : "Appraisal Meeting",
   startDate : "2016-10-04 16:00:00",
   endDate : "2016-10-04 17:00:00",
   location : "Martha's office",  //在双引号界定的字符串中包含表示所有格的撇号
   importance: 1,
```

```
        notes : 'Don\'t forget the portfolio!' //在单引号界定的字符串中包含
                                               //表示缩略的撇号
};
```
35

请注意如何处理属性 notes 中的撇号。撇号之前的反斜杠符起到以下作用：避免 JavaScript 将撇号看作表示字符串结尾的单引号。这个反斜杠称为转义字符，在程序输出时不会显示，如下所示：

```
event.notes = 'Don\'t forget the portfolio!';
```

以上代码的输出结果：Don't forget the portfolio!

使用双引号界定字符串时，转义字符能够显示该字符串内包含的双引号，如下所示，两个转义字符后面的双引号都会显示出来。

```
var story = "She looked at me. \"What did you say?\" she asked.";
```

以上代码的输出结果：

```
She looked at me. "What did you say?" she asked.
```

JavaScript 也使用反斜杠转义字符来标明特殊字符，例如，\t 为 Tab 键，\r 为回车键。

日历中包含大量的事件对象。如果所创建的对象具有相似的结构，是否能够精简这个创建过程？在第 9 章，读者将学习使用构造函数来解决上述问题。

3.5.3 天气怎么样?

在线气象服务可以为我们的程序提供天气数据。这些数据通常使用 JSON 格式（JavaScript Object Notation 见第 20 章），这些数据包含许多详细的属性，与本章中学到的对象非常相似。以下代码显示了由在线气象服务提供的定位数据。



代码清单 3-14 天气应用中的定位
(http://jsbin.com/diguhe/edit?js,console)

```
var location = {
    "city"          :  "San Francisco",
    "state"         :  "CA",
    "country"       :  "US",
    "zip"           :  "94101",
    "latitude"      :  37.775,
    "longitude"     :  -122.418,
    "elevation"     :  47.000
};
```

在以上代码中，将“键”放在双引号内。尽管在以前的例子中还没有看到将“键”放在引号内的情形，但事实上，JavaScript 常常会将“键”（属性名称）放在引号中，单引号或双引号都可以。尤其是当属性名称不满足在第 2 章中提到的变量名称的要求时，引号是必不可少的。本书将在第 10 章更详细地介绍如何使用此类属性的名称。JSON 格式规定了如何在互

联网上将 JavaScript 对象数据作为文本进行传输。JSON 格式要求所有“键”（属性名称）都必须放在双引号中。在编程工作中，有时候会遇到诸如 JSON 格式的强制要求，因此许多程序员的经验之谈是索性把“键”全部放在引号内，以免造成程序的不一致和潜在错误。 36

以上代码中还将“键-值对”中的冒号进行了整齐排列。意义何在？将其与本节中的其他代码进行比较，是不是感觉这样整齐排列的代码更加容易阅读？这确实是很多程序员长期以来形成的格式习惯，你可以效仿，当然也可以摒弃。

3.5.4　一个小测验

经常自测是一种很好的学习方法。测验应用程序可以发现问题，并将答案作为对象的属性，如以下代码所示。



代码清单 3-15　测验应用程序中的问题和答案 (http://jsbin.com/damoto/edit?js,console)

```
var questionAndAnswer = {
   question : "What is the capital of France?",
   answer1  : "Bordeaux",
   answer2  : "F",
   answer3  : "Paris",
   answer4  : "Brussels",
   correctAnswer  : "Paris",
   marksForQuestion  :  2
};
```

测验应用程序可能包括多种问题类型，上述代码是一道多项选择题。其实，针对每种问题类型都有固定的呈现形式，在呈现这些类似结构的数据时，模板是一种好方法，我们将在第 19 章详细讨论。

3.5.5　创建自己的程序

完成以上几个示例之后，你自己也跃跃欲试了吧？想要在程序中设计什么样的对象来代表你的实体内容？去 JS Bin 吧！在那里，可以创建对象并将其属性显示在控制台上。去本书的在线论坛吧！在那里，可以分享作品，还可以与同行讨论问题。论坛的网址是 https://forums.manning. com/forums/get-programming-with-javascript。

3.6　游戏 The Crypt——玩家对象

在本节中，要把我们刚刚学到的 JavaScript 有关对象的相关知识应用到开发游戏中去。图 3-5 显示，目前你在游戏中选中了一个玩家对象。在第 2 章中，考虑到游戏 The Crypt 中需要为玩家存储信息，我们使用了以下变量来呈现一位玩家：

```
playerName = "Kandra";
playerHealth = 50;
playerPlace = "The Dungeon of Doom";
```

37
```
playerItems = "a rusty key, The Sword of Destiny, a piece of cheese";
```

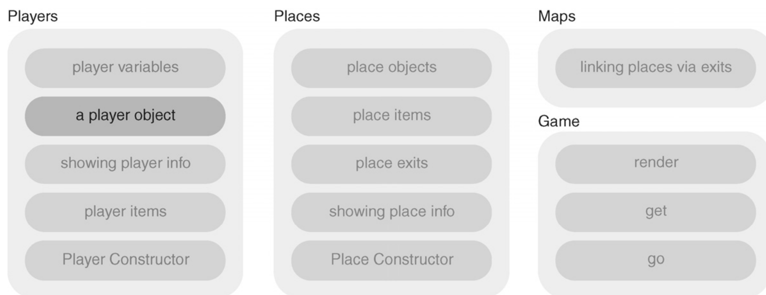


图 3-5　The Crypt 中的游戏元素

针对多个玩家，可以使用变量前缀，例如 player1Name，player2Name 等,为游戏中的每个玩家复制这些变量。显然，这种使用 JavaScript 对象将每位玩家的所有信息进行打包分组的方法非常简洁明了。以下代码清单 3-16 显示了如何将一位玩家表示为一个对象，并在控制台上显示对象的属性。输出如下：

```
> Kandra
> Kandra is in The Dungeon of Doom
> Kandra has health 50
> Items: a rusty key, The Sword of Destiny, a piece of cheese
```



代码清单 3-16　将一位玩家作为一个对象 (http://jsbin.com/qelene/edit?js,console)

```
var player;
player = {
   name: "Kandra",
   health: 50,
   place: "The Dungeon of Doom",
   items: "a rusty key, The Sword of Destiny, a piece of cheese"
};
console.log(player.name);
console.log(player.name + " is in " + player.place);
console.log(player.name + " has health " + player.health);
console.log("Items: " + player.items);
```

以上代码的最后 4 行仅用于显示玩家的信息。每次想要显示玩家的信息都必须重复这些代码，这似乎有点麻烦。如果能够一次性编写这些代码，然后根据需要多次调用，那就太好了！

非常幸运的是，确实可以这样做！在 JavaScript 中可以定义函数（functions）。函数就是一组执行某项任务的 JavaScript 代码。函数的功能非常强大，上文中提到两项任务，重复多
38 次创建多个玩家对象以及显示玩家的多个属性，会因为使用函数而变得简单。读者将在后续的四章中详细了解 JavaScript 函数。

3.7　本章小结

- 将相关的变量分组。
- 将对象定义为若干属性的集合，这些属性用花括号括起来，各个属性之间用逗号分隔：

```
var player = { name : "Hadfield", location : "The ISS" };
```

- 对于每个属性，使用一个键-值对，键和值之间用冒号分隔：

```
name："Hadfield"
```

- 使用圆点运算符访问属性的值。如果将一个对象分配给一个变量，使用圆点运算符将属性名称连接到变量名称之后：

```
player.name
```

- 在表达式中可以像使用变量一样使用属性：

```
console.log（player.name +"is in"+ player. location）;
```

- 使用赋值运算符 = 为属性赋值：

```
player.location ="On a space walk";
```

- 根据需要，可以随时向现有对象添加新属性：

```
player.oxygen = 96;
```
39

第 4 章　函数：按需执行代码

本章内容包括：

- 编写有函数的指令
- 定义函数——明确说明代码是按需执行任务
- 调用函数——按需执行代码
- 减少代码中的重复
- 使程序更易于阅读和更新

《JavaScript 开发实战》的一个重要主题就是通过良好的组织来加强对程序的复杂性管理。在本书的第 2 章中，我们学会将信息存储在变量中，选取合适的变量名有助于理解该变量在程序中的用途。在本书的第 3 章中，我们学会将变量分组成为对象的属性，这样可以将对象看作一个整体，只有在需要时才深入其细节。在本章我们将看到另一种组织代码的方法，它可以避免重复，它就是我们将要学习的函数。

4.1　重复问题

随着程序越来越长、越来越复杂，我们发现有些部分的代码非常类似，相互之间只有微
40 小的差别，尤其是一些经常用到的任务，例如显示文本、为图像设置动画或者将数据保存到数据库。现在我们就来关注这些重复的代码段，它们就是函数的主要素材。

函数就是只需一次编写但是可以多次使用的代码。本书第 4.2 节将讨论如何创建函数，在这里首先呈现几个重复的示例。

4.1.1　将对象的属性作为文本进行显示

程序使用对象和变量来存储各种信息——个人资料、帖子、文档和照片，首先对其命名，然后把它们存储在计算机的某个地方。我们常常需要向用户显示这些信息，例如，针对一些表示电影的对象，常常需要将每部电影的信息显示在控制台上，如图 4-1 所示。



图 4-1　在 JS Bin 控制台上显示一部电影的信息

为了实现图 4-1 中的输出任务，仅仅针对一部电影，程序就 5 次调用 console.log。如下所示：



代码清单 4-1　在控制台上显示一个对象的多个属性 (http://jsbin.com/besudi/edit?js, console)

```
var movie1;
movie1 = {
   title: "Inside Out",
   actors: "Amy Poehler, Bill Hader",//创建一个对象 movie1
   directors: "Pete Doctor, Ronaldo Del Carmen"
};
console.log("Movie information for " + movie1.title);
console.log("------------------------------");
console.log("Actors: " + movie1.actors);  //调用 console.log 5 次，将对象
                                          //movie1 的各个属性显示在控制台上
console.log("Directors: " + movie1.directors);
console.log("------------------------------");
```

41

如果每次要显示一部电影信息时都必须写以上五行代码，这完全是重复性工作。如果想要改变显示的信息，就必须对其在代码中出现的所有地方都做出修改，并且确保修改前后一致。

下面的代码清单展示了针对三部电影的重复编码。



代码清单 4-2　将若干相似对象的属性显示在控制台上 (http://jsbin.com/gewegi/edit?js,console)

```
console.log("Movie information for " + movie1.title);
console.log("------------------------------");
console.log("Actors: " + movie1.actors);
console.log("Directors: " + movie1.directors);
console.log("------------------------------");
console.log("Movie information for " + movie2.title);
console.log("------------------------------");
console.log("Actors: " + movie2.actors);
console.log("Directors: " + movie2.directors);
console.log("------------------------------");
console.log("Movie information for " + movie3.title);
console.log("------------------------------");
console.log("Actors: " + movie3.actors);
console.log("Directors: " + movie3.directors);
console.log("------------------------------");
```

需要显示的电影可能不止三部，需要显示的电影信息也可能会更多。例如，我们需要将 information 更改为 info，就必须确保找到 information 出现的所有地方，再将其全部更改为 info。

以上三组（每一组包含 5 个）语句几乎相同。唯一的差异就是显示具体哪一部电影的属性，是 movie1，是 movie2，还是 movie3。如果我们能够定义一组语句，并要求 JavaScript 在需要时调用这一组语句，那就太好了！这就是函数的意义所在！

下一节还会有一个重复编码的示例。（别担心，当我们对重复编码忍无可忍之时，就不必再忍了！函数会来拯救我们！）

4.1.2　加税算出总成本

将税额（tax）与价格（price）相加是一个非常简单又常见的任务。我们需要将税额与价格相加，以计算出总成本（totalcost），如下所示：

```
>price = $140
>tax @ 15% = $21
>totalcost = $161
```

42 以下代码清单是一个为三宗不同交易加税的程序。两个运算符*和/分别执行乘法和除法。



代码清单 4-3　加税算出总成本
(http://jsbin.com/kawocu/edit?js,console)

```
var sale1;
var sale2;
var sale3;
sale1 = { price: 140, taxRate: 15 };
sale2 = { price: 40, taxRate: 10 };
sale3 = { price: 120, taxRate: 20 };
sale1.tax = sale1.price * sale1.taxRate / 100;  //这三个计算的结构完全相同，运
                                                //算符*和/分别执行乘法和除法
sale2.tax = sale2.price * sale2.taxRate / 100;
sale3.tax = sale3.price * sale3.taxRate / 100;
sale1.total = sale1.price + sale1.tax;  //这三个计算的结构也完全相同
sale2.total = sale2.price + sale2.tax;
sale3.total = sale3.price + sale3.tax;
console.log("price = $" + sale1.price);
console.log("tax @ " + sale1.taxRate + "% = $" + sale1.tax);
console.log("totalcost = $" + sale1.total);
console.log("price = $" + sale2.price);
console.log("tax @ " + sale2.taxRate + "% = $" + sale2.tax);
console.log("totalcost = $" + sale2.total);
console.log("price = $" + sale3.price);
console.log("tax @ " + sale3.taxRate + "% = $" + sale3.tax);
console.log("totalcost = $" + sale3.total);
```

重复这么多次简直太烦人了！不仅 console.log 代码在多次重复，计算的结构也在重复。我们实在想要有一组代码，每次需要时，拿来就可以用。放心，下面我们就会学到一个好办法来解决这个重复问题。让我们一起来学习函数吧！

4.2　定义函数和调用函数

正如一个对象是若干个属性的集合，一个函数就是一组语句或指令的集合。函数有助于避免重复，使代码更有条理，并且更易于更新和维护。我们知道，给变量和对象取一个合适的名字可以使代码容易理解。同理，命名良好的函数也会增强程序的可读性。如果发现某些函数在程序中被多次使用，并且在其他程序中也非常有用，就可以创建一个实用函数库以供更多的项目使用。

在上一节中，我们看到两个程序示例，其中相同结构的代码反复出现。为了减少代码膨胀，可以使用以下代码来替换上述重复部分：

```
showMovieInfo( );
showCostBreakdown( );
```

43

使用这两个函数，showMovieInfo 和 showCostBreakdown，应该产生与代码清单 4-2 和 4-3 等效的输出。这两个函数也应该能够在需要时被多次调用。下面我们来看看这样的魔法代码如何做到随叫随到！

4.2.1　定义新函数

定义一个函数包含以下元素，如图 4-2 所示：

- 关键字 function
- 圆括号，()
- 花括号之间的代码块，{ }

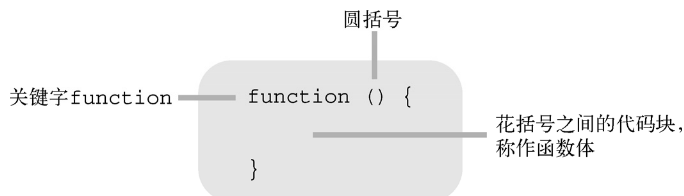


图 4-2　函数定义所包含的元素

花括号之间的代码块就是在任何时候使用该函数时所执行的功能，这些代码块也称为函数体。以下是函数定义的大致模样：

```
function () {
    //这里是要执行的代码块
}
```

一旦定义了一个函数，就可以将它赋值给一个变量。在下一个代码清单中定义了一个函数，在控制台上显示“Hello World!”，并且还将该函数赋值给变量 sayHello。



代码清单 4-4　一个简单的函数定义和赋值
(http://jsbin.com/tehixo/edit?js,console)

```
var sayHello; //声明变量
```

```
sayHello = function ( ) {       //定义函数，并将该函数赋值给变量 sayHello
   console.log("Hello World!");//将代码放在花括号之间，每当运行函数时就执行该代码
44 };
```

我们可以看到，以上代码中的函数定义由以下几部分组成：函数关键字 function，空的圆括号和函数体代码块。这个函数体只有一条语句：console.log(“HelloWorld!”)；就这样，我们定义了一个函数，以后就可以使用该函数。每次运行该函数，都会执行函数体中的代码，程序的实际运行情况在 4.2.3 节中可以看到。

以下代码清单 4-5 是将有关函数定义后将它们赋值给变量的更多示例。



代码清单 4-5　有关函数定义和赋值的另外两个示例
(http://jsbin.com/xezani/edit?js,console)

```
var findTotal;
var displayMenu;
findTotal = function ( ) {
   result = number1 + number2;     //在程序的其他地方必须已经声明过变量 result，
                                   //number1 和 number2
};
displayMenu = function ( ) {
   console.log("Please choose an option:");
   console.log("(1) Print log");
   console.log("(2) Upload file");
   console.log("(9) Quit");
};
```

4.2.2　函数表达式和函数声明

在上述例子中，一直在使用函数表达式（function expressions）来定义函数，并使用赋值运算符将函数赋值给变量，如下所示：

```
var findTotal = function () { … }; //函数表达式为粗体
```

除了以上使用函数表达式的方法，还可以使用函数声明（function declaration），就是在定义函数的同时为该函数声明一个名称，如下所示：

```
Function findTotal () { … }   //为该函数声明一个名称
```

采用以上两种方法定义函数基本上等效，当然也有一些细微的差异，但本书对此差异不做详述。在本书第 1、2 章都采用了函数表达式的方法，目的在于强调函数与对象、数组三者在创建和赋值方面的相似性，如下所示：

```
var numOfDays = 7;//将一个数字赋值给一个变量
var player = { … };//创建一个对象，并将其赋值给一个变量
var findTotal = function () { … }; //创建一个函数，并将其赋值给一个变量
45 var items = []; //创建一个数组，并将其赋值给一个变量(见第 8 章)
```

目前，读者暂时不必考虑函数声明，更不必考虑函数声明和函数表达式之间的差别。当

你在本书的第三部分中再次遇到函数声明这个语法时，就可以轻松驾驭了。

仅仅依靠定义函数，我们还是无法实现以下功能：在控制台上显示“Hello World!”、计算总数或显示菜单。我们需要一种方法来命令函数执行其函数体中的指令列表。

4.2.3　使用函数

一旦将函数赋值给一个变量，每当要执行函数体中的语句时，就在该变量名后面加上一个圆括号（）即可，如下所示：

```
sayHello( );
findTotal( );
displayMenu( );
```

运行函数也称为调用函数。在以下代码清单 4-6 中，三次调用函数 sayHello，就会三次显示字符串“Hello World!”，如下所示：

```
>Hello World!
>Hello World!
>Hello World!
```



代码清单 4-6　三次调用函数 sayHello (http://jsbin.com/vozuxa/edit?js,console)

```
var sayHello;
sayHello = function ( ) {      //定义函数并将其赋值给一个变量
   console.log("Hello World!");
};
sayHello( );         //通过在变量名后面加圆括号的方法来调用函数
sayHello( );
sayHello( );         //多次调用函数
```

下一个代码清单使用函数 findTotal 来更新变量 result，然后在控制台上显示整个算式：

```
> 1000 + 66 = 1066
```



代码清单 4-7　使用函数 findTotal 来显示整个算式 (http://jsbin.com/hefuwa/edit?js,console)

```
var number1 = 1000;
var number2 = 66;
var result;
varfindTotal;
findTotal = function ( ) {
   result = number1 + number2;
};
findTotal();
console.log(number1 + " + " + number2 + " = " + result);
```

代码清单 4-8 显示了如何调用函数 displayMenu 来显示菜单。

46

```
>Please choose an option:
> (1) Print log
> (2) Upload file
> (9) Quit
```



代码清单 4-8　显示菜单
(http://jsbin.com/cujozo/edit?js,console)

```
var displayMenu;
displayMenu = function ( ) {
    console.log("Please choose an option:");
    console.log("(1) Print log");
    console.log("(2) Upload file");
    console.log("(9) Quit");
};
displayMenu();
```

在读者看来，使用空的圆括号作为调用函数的符号可能会有些奇怪。但是，在第 5 章就会看到，圆括号内并不总是空的。

定义　对于那些喜欢术语的人来说，可以在这里多学一招：当调用一个函数时，添加到变量后面的圆括号（）称作函数调用操作符。

4.2.4　使用函数的步骤

表 4-1 罗列了定义函数和调用函数的步骤。

表 4-1　定义函数和调用函数

47

行　　为	代　　码	注　　释
声明变量	var sayHello;	设置一个在程序中使用的变量名称
定义函数	function () { console.log("Hello World!"); }	在此阶段，函数体内的代码还不能执行
将函数赋值给变量	sayHello = function () { console.log("Hello World!"); };	将函数赋值给变量就意味着为程序能够调用该函数发了一张通行证
调用函数	sayHello();	函数体内的代码得以执行
按需多次调用函数	sayHello(); sayHello(); sayHello();	函数体内的代码可以多次执行

4.3　减少重复

在代码清单 4-2 和 4-3 中，我们看到，在没有使用函数的情况下，程序中存在大量重复的代码块。通过使用函数，可以大大减少上述重复现象。

4.3.1　函数可用于将对象的属性作为文本进行显示

重新返回到代码清单 4-2 中的代码，现在我们用函数来简化对电影信息的显示。将显示

电影信息的代码一次性写入一个函数，然后就可以根据需要轻松地调用该函数。



代码清单 4-9　使用函数来显示对象的属性
(http://jsbin.com/toqopo/edit?js,console)

```
var showMovieInfo;
showMovieInfo = function ( ) {
   console.log("Movie information for " + movie.title);
   console.log("------------------------------");
   console.log("Actors: " + movie.actors);
   console.log("Directors: " + movie.directors);
   console.log("------------------------------");
};
```

以上代码中，将一个新函数分配给变量 showMovieInfo，然后通过该变量名称后跟圆括号，showMovieInfo()来调用这个函数，如代码清单 4-10 所示，最终在控制台上得到以下输出，与图 4-1 所示目标完美匹配。 48

```
> Movie information for Inside Out
> ------------------------------
> Actors: Amy Poehler, Bill Hader
> Directors: Pete Doctor, Ronaldo Del Carmen
> ------------------------------
```



代码清单 4-10　调用函数 showMovieInfo
(http://jsbin.com/menebu/edit?js,console)

```
var movie1;
var showPlayerInfo;
var movie;
movie1 = {
   title: "Inside Out",
   actors: "Amy Poehler, Bill Hader",
   directors: "Pete Doctor, Ronaldo Del Carmen"
};
showMovieInfo = function ( ) { //定义函数，在此阶段，函数体内的代码还不能执行
   console.log("Movie information for " + movie.title);
   console.log("------------------------------");
   console.log("Actors: " + movie.actors);
   console.log("Directors: " + movie.directors);
   console.log("------------------------------");
};
movie = movie1;
showMovieInfo( );  //调用函数，执行函数体内的代码
```

在以上代码中，通过使用变量 movie 并调用函数 showMovieInfo，可以改变函数中电影的信息。代码清单 4-11 显示了如何切换电影，将三部电影的信息显示在控制台上：

```
>Movie information for Inside Out
> -----------------------------
>Actors: Amy Poehler, Bill Hader
>Directors: Pete Doctor, Ronaldo Del Carmen
> -----------------------------
>Movie information for Spectre
> -----------------------------
>Actors: Daniel Craig, Christoph Waltz
>Directors: Sam Mendes
> -----------------------------
>Movie information for Star Wars: Episode VII - The Force Awakens
> -----------------------------
>Actors: Harrison Ford, Mark Hamill, Carrie Fisher
>Directors: J.J.Abrams
```

49

```
> -----------------------------
```



代码清单 4-11　多个对象使用同一个函数
(http://jsbin.com/mavutu/edit?js,console)

```
var movie1;
var movie2;
var movie3;
var movie;          //声明一个变量 movie 来存放当前显示的电影
var showMovieInfo;
movie1 = {
   title: "Inside Out",
   actors: "Amy Poehler, Bill Hader",
   directors: "Pete Doctor, Ronaldo Del Carmen"
};
movie2 = {
   title: "Spectre",
   actors: "Daniel Craig, Christoph Waltz",
   directors: "Sam Mendes"
};
movie3 = {
   title: "Star Wars: Episode VII - The Force Awakens",
   actors: "Harrison Ford, Mark Hamill, Carrie Fisher",
   directors: "J.J.Abrams"
};
showMovieInfo = function ( ) {     //在函数 showMovieInfo 中使用变量 movie
   console.log("Movie information for " + movie.title);
   console.log("------------------------------");
   console.log("Actors: " + movie.actors);
   console.log("Directors: " + movie.directors);
   console.log("------------------------------");
};
movie = movie1;          //将 movie1 赋值给变量 movie，为显示该电影做准备
```

```
showMovieInfo( );
movie = movie2;          //切换准备显示的电影：movie2 和 movie3
showMovieInfo( );
movie = movie3;
showMovieInfo( );
```

4.3.2　加税并显示总成本的函数

以下代码清单 4-12 显示了一个为销售增加税额并显示每笔交易总成本的函数。原先大段大段重复的代码都被两个函数 calculateTax 和 displaySale 替代，针对每个销售对象依次调用这两个函数，就可以得到以下输出：

```
>price = $140
>tax @ 15% = $21
>totalcost = $161
>price = $40
>tax @ 10% = $4
>totalcost = $44
>price = $120
>tax @ 20% = $24
>totalcost = $144
```

50

正如 JS Bin 上所有其他代码清单一样，在程序的下面可以看到 Further Adventures（进阶练习），在这里，对程序编写提出进一步优化的思考。针对代码清单 4-12 的优化建议就是尽量减少函数 calculateTax 和 displaySale 一起被调用的重复次数。尽管二者执行两个不同的功能，是否可以避免每一笔交易都要调用这两个函数？如果你可以连接互联网，请单击代码清单 4-12 的链接，马上前往 JS Bin 去迎接挑战！如果你只是阅读本书，那实际上就是远离了技术。记住，JS Bin 才是你编程的纸和笔！本书的在线平台能够为大多数问题提供解决方案，链接如下：www.room51.co.uk/books/getprogramming/listings.html。



代码清单 4-12　使用函数计税并显示
(http://jsbin.com/raqiri/edit?js,console)

```
var sale1;
var sale2;
var sale3;
var sale;     //先声明一个变量 sale，在后面的函数中将会使用该变量
var calculateTax;
var displaySale;
sale1 = { price: 140, taxRate: 15 };
sale2 = { price: 40, taxRate: 10 };
sale3 = { price: 120, taxRate: 20 };
calculateTax = function ( ) {
   sale.tax = sale.price * sale.taxRate / 100; //这里使用了变量 sale，在后面
                                               //的函数 displaySale 中还会使用该变量
```

```
        sale.total = sale.price + sale.tax;
    };
    var displaySale = function ( ) {
        console.log("price = $" + sale.price);
        console.log("tax @ " + sale.taxRate + "% = $" + sale.tax);
        console.log("totalcost = $" + sale.total);
    };
    sale = sale1; //将 sale1 赋值给 sale，然后就可以调用下面两个函数来更新并显示计税
    calculateTax( );
    displaySale( );
    sale = sale2;
    calculateTax( );
    displaySale( );
    sale = sale3;
    calculateTax( );
    displaySale( );
```

以上代码在定义这两个函数时都使用了变量 sale，在访问对象的属性 sale.price 和
51 sale.taxRate 时也使用了变量 sale。在调用这两个函数之前，函数体中的代码是不能运行的，一直要等到程序调用这两个函数，函数体中的代码才得以运行。届时，程序会将某个销售对象赋值给变量 sale。

函数名称 calculateTax 和 displaySale 可使代码清单 4-12 更易于阅读，具体内容将在第 4.4 节详述。

4.4 使代码易于阅读和更新

随着程序变得越来越长、越来越复杂，我们可以采用对对象和函数适当命名的方式将程序分成若干部分，以此来降低程序的复杂性。这样一来，当其他人阅读该代码时，就可以跟上编写者的思路，理解程序每个部分的意思和整体目的。

请读者阅读下面的代码片段，我相信，尽管读者不知道该函数如何工作的细节，应该也能够理解其大致意图。

```
...
var balance = getAccountBalance( );
displayBalance( );
addInterest( )
addBonus( );
setAccountBalance( );
displayBalance( );
...
```

如上例所示，每个函数都应该具有单一、明确的目标。如果需要研究某一个函数，就应该首先找到在何处定义了该函数。下面，我们来看一个更新函数的示例。

4.4.1　更新 showMovieInfo 函数

在前文代码清单 4-11 中，创建了一个函数 showMovieInfo 来显示某个电影对象的信息，像那样将显示代码块集成到一个函数里是非常好的，但是当多个电影的信息都混在一起显示在控制台上，就很难挑选出关于某个特定电影的独特信息。为了解决这个问题，我们可以通过添加空行的办法使每一部电影的信息得以清楚地显示，如下所示：

```
>Movie information for Inside Out
> ------------------------------
>Actors: Amy Poehler, Bill Hader
>Directors: Pete Doctor, Ronaldo Del Carmen
> ------------------------------
>
>Movie information for Spectre
> ------------------------------
>Actors: Daniel Craig, Christoph Waltz
>Directors: Sam Mendes
> ------------------------------
>
>Movie information for Star Wars: Episode VII - The Force Awakens
> ------------------------------
>Actors: Harrison Ford, Mark Hamill, Carrie Fisher
>Directors: J.J.Abrams
> ------------------------------
>
```

58

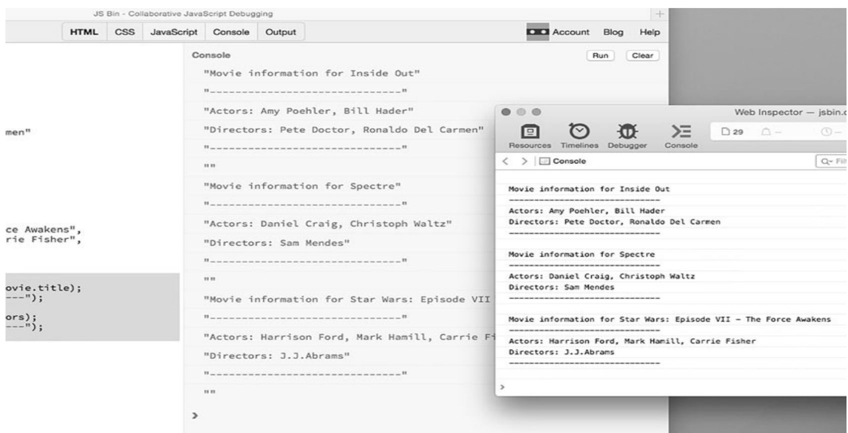
因为代码清单 4-11 中所有的显示代码都集成在函数 showMovieInfo 里，因此可以直接在其函数体内部添加额外的一行来调用 console.log，从而显示一个空白行，如代码清单 4-13 所示。



代码清单 4-13　更新显示函数来增加一个空白行 (http://jsbin.com/cijini/edit?js,console)

```
showMovieInfo = function () {
   console.log("Movie information for " + movie.title);
   console.log("------------------------------");
   console.log("Actors: " + movie.actors);
   console.log("Directors: " + movie.directors);
   console.log("------------------------------");
   console.log("");     //添加额外的一行来调用 console.log，从而创建一个空白行
};
```

请仔细看！因为 JS Bin 控制台上用引号""来显示字符串的左右端，所以并没有真正看到空白行，而是看到一个中间内容为空的引号""。但是如果使用浏览器（参见在线指南 www.room51.co.uk/guides/browser-consoles.html），就会看到预期的空白行，如图 4-3 所示。



53 图 4-3　JS Bin 控制台显示带引号的空字符串，而浏览器显示则无引号

如果以上例子来自一个更大的程序，假如我们没有让玩家的信息按照一定逻辑嵌套在函数里，那么当我们要对某些内容进行修改时，就必须仔细检查所有代码，才能找到应该进行更改的代码行。文本编辑器和开发环境虽然能够为我们的修改提供一些帮助工具，但并不能做到万无一失，最终程序有可能就是因为这些未纠正的代码而无法运行。你可以自己试一试，刚开始程序很短，似乎一切都很好，然后随着程序变长，问题就来了。为了避免这样的噩梦，还是选择使用函数吧！

4.5　游戏 The Crypt——显示玩家的信息

在本节中，将要把读者在本章所学 JavaScript 函数的知识应用到游戏 The Crypt 中。如图 4-4 所示，本节的重点是通过使用函数显示玩家的信息。

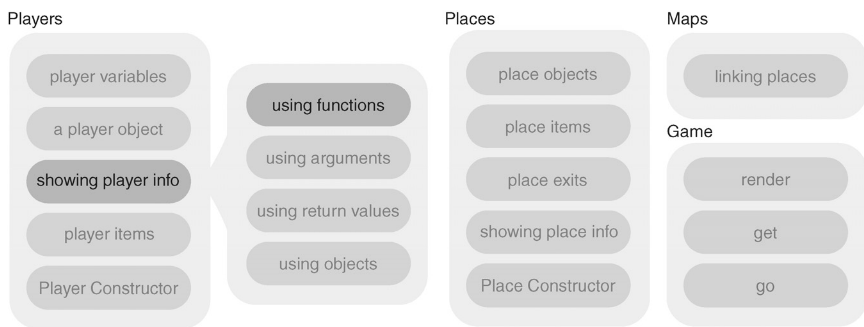


图 4-4　The Crypt 中的游戏元素

在第 3 章中，我们看到如何将一个玩家的信息分组构成一个 JavaScript 对象。我们使用花括号创建对象，并使用键-值对为对象设置属性，如下所示：

```
var player;
player = {
    name: "Kandra",
```

```
    health: 50,
    place: "The Dungeon of Doom"
};
```

一旦将新对象赋值给变量 player 后，即可使用圆点运算符来获取该变量的属性值，并将该玩家的信息显示在控制台上：

```
console.log(player.name + " is in " + player.place);
```

54

在游戏中，可能需要多次显示多个玩家的信息，这就需要我们使用本章的函数新知识来更加高效地显示信息。函数既然是按需代码，那就意味着我们在需要时可以通过调用函数来执行代码。

4.5.1　用函数显示玩家的信息

在 4.4.1 节，代码清单 4-13 中的函数 showMovieInfo 看起来正是我们所需要的这种函数。showMovieInfo 显示关于电影的信息，以下代码清单 4-14 中的函数 showPlayerInfo 显示的是与此类似的游戏玩家信息，在控制台上输出如下：

```
>Kandra
> ------------------------------
>Kandra is in The Dungeon of Doom
>Kandra has health 50
> ------------------------------
>
>Dax
> ------------------------------
>Dax is in The Old Library
>Dax has health 40
> ------------------------------
>
```



代码清单 4-14　用函数显示玩家的信息
(http://jsbin.com/mafade/edit?js,console)

```
var player1;        //声明变量
var player2;
var player;
var showPlayerInfo;
player1 = {         //使用花括号和键-值对创建对象，并将其赋值给变量
    name: "Kandra",
    place: "TheDungeonofDoom",
    health: 50
};
player2 = {
    name: "Dax",
    place: "TheOldLibrary",
    health: 40
```

```
    };
    showPlayerInfo = function () {      //定义一个函数来显示玩家的属性。当调用该函
                                        //数时，执行函数体的代码
        console.log(player.name);
        console.log("----------------------------");
        console.log(player.name + " is in " + player.place);
        console.log(player.name + " has health " + player.health);
        console.log("----------------------------");
        console.log("");
55  };
    player = player1;  //将一个玩家对象player1 赋值给玩家变量player，以便函数可
                       //以访问到它
    showPlayerInfo();  //调用函数来显示玩家对象的属性
    player = player2;  //将另一个玩家对象player2 赋值给玩家变量player
    showPlayerInfo();  //再次调用函数，这是必需的代码
```

非常棒！现在只需要简单地调用一下函数就可以显示玩家的信息了。但是依然还有点儿麻烦，我们必须一次又一次地把每一个不同的玩家赋值给变量 player。如果能以某种方式对函数直接说，"显示 player1 的信息"或"显示 player2 的信息"，那就更好了！关于如何向函数传入和传出信息，本书将在后续三章中详细论述。我们的目标是使程序更加灵活、高效、可复制！

4.6 本章小结

- 函数是一次编写但可以多次使用的代码块。 每个函数应该具有明确、单一的目的。
- 定义函数需要使用关键字 function、圆括号以及圆括号中间的代码（函数体）：

```
function ( ) {
    // 函数体内的语句
};
```

- 使用等号将函数赋值给变量，等号也称为赋值运算符：

```
showPlayerInfo = function () { … };
```

- 一旦将函数赋值给一个变量，就可以调用该函数。调用函数的方法是在变量名后面添加圆括号：

```
addTax () ;
showPlayerInfo () ;
evadeRaptor () ;
```

- 寻找重复性代码，或者结构相同，仅仅值或变量有轻微变化的代码段，然后将重复的代码放到一个函数中。

56 - 为函数起一个清楚明了的名称，使程序更容易阅读和维护。

第 5 章　参数：将数据传递给函数

本章内容包括：

- 使用形参（parameters）定义函数，为传入数据做好准备
- 使用实参（arguments）调用函数，为函数传入数据

函数是组织程序代码的一个重要手段，只需要一次性编写函数代码，之后就可以多次使用该函数。在本章之前，函数都是和它所赋值的变量绑定在一起使用。从本章开始，我们可以自由地设置函数，让函数命名自己的变量，并且程序还可以将需要的数据传递给函数。

5.1　函数重用

在本章之前，我们使用函数时总是依赖于函数之外的一个变量，而对该变量的声明和赋值是在程序的其他地方。在以下代码清单 5-1 中，函数 showMessage 依赖于一个名为 message 的变量，而对 message 变量的声明并不在函数定义的过程中，函数定义和变量声明分别存在于程序的不同位置。



代码清单 5-1　函数依赖于函数之外的另一个变量 (http://jsbin.com/taqusi/edit?js,console)

```
var message;
var showMessage;
message= "It's full of stars!";    //将欲显示的字符串赋值给变量 message
showMessage = function () {
   console.log(message); //在函数体内使用变量message，该变量在函数之前已经定义
};
showMessage();      //调用函数 showMessage，在该函数中使用变量 message 当前的值
```

在 showMessage 函数定义中，使用了一个名为 message 的变量。使用一个变量的前提是已经声明过该变量，然后才能为函数所使用。如果该变量的名称被修改，那么该函数就不能再使用这个变量了。例如变量 message 被重新命名为 msg，如以下代码清单 5-2 所示，在 JS Bin 上运行程序，就会得到一个错误信息：“Reference error: Can’t find variable: message”（不同浏览器显示的错误信息可能会略有不同）。



代码清单 5-2　更改变量名导致函数无法正常运行 (http://jsbin.com/yaresa/edit?js,console)

```
var msg;
```

```
var showMessage;
msg= "It's full of stars!";          //将欲显示的字符串赋值给变量 msg
showMessage = function () {
   console.log(message);             //在函数体内使用一个名为 message 的变量
};
showMessage();                       //调用函数 showMessage，产生了错误
```

函数体内的指令确实可以使用在程序中其他位置已经声明过的变量，但是这样做就造成了函数与外部变量之间的绑定关系。当该变量发生变化时，函数也必须随之做出相应的变化，否则就会出错，显然这不是使用函数的理想状态。下面介绍一种更好的方法，在这种方法中，只有在调用函数时，才将函数需要的信息传递给该函数。以上做法有助于避免函数需要的变量发生错误命名、丢失、删除或者在程序的其他部分更改等情况，这样做还有助于使程序更加易读，而且在出错时更容易找到错误。

如同摇滚歌星在出场之前一定要在更衣室内隆重着装,我们一直以来所使用的函数中的变量在使用之前也必须事先声明，这样傲慢的函数并不讨人喜欢。我们更喜欢有亲和力的歌手，他们随时随地都乐意出场表演。通过去除函数和变量之间的上述绑定关系，可以使函数
58 更加独立，便于在不同程序间多次使用，而不会引发错误。

那么，如何去除函数和变量之间的绑定关系？

5.2 将信息传递给函数

将信息传递给函数分两步进行：定义函数和调用函数。

（1）定义函数：设置变量名称，该变量名可以称为形参，作用是为下一步调用函数做好准备。

（2）调用函数：为以上在步骤（1）中命名的形参传入数据。

在本章（甚至在整本书）的所有示例中，都没有关于如何设置以上参数的练习。但是，在 JS Bin 上，每个代码清单的 Further Adventures 部分都可以看到关于设置参数的练习。

5.2.1 将实参传递给函数

现在，我们使用空的圆括号来定义函数。为了在调用函数时能够将信息传递到该函数，可以将该信息放在圆括号之间，如下所示：

```
showMessage("It's full of stars!");
showPlayerInfo("Kandra");
getMovieActors("The Hobbit");
square(12);
```

在以上示例中，我们向上述四个函数分别传递了一些信息供其函数代码使用。每一个包括在圆括号中的值称为实参。上述每一个函数都包含一个实参，分别是“It’s full of stars!”、“Kandra”、“The Hobbit”和 12。

在使用以上圆括号中的信息之前，必须确保函数已经做好准备来接受这些信息。我们可以在定义函数时添加形参，形参表明该函数期望在调用该函数时给它传递一些信息，如图 5-1 所示。

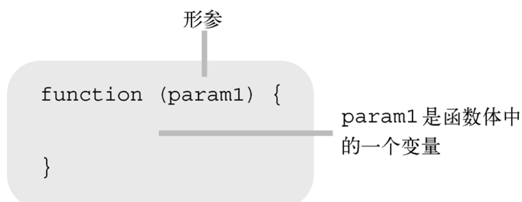


图 5-1　定义函数时将形参放在圆括号内 59

对于上述示例中的 4 个函数，其函数定义大致如下所示：

```
showMessage = function (message) { … };
showPlayerInfo = function (playerName) { … };
getMovieActors = function (movieTitle) { … };
square = function (numberToSquare) { … };
```

上述每个函数定义中都包括一个参数，以粗体显示。该参数是一个变量，只能在函数体内部使用。

对于前文代码清单 5-1 中函数 showMessage 内容的更新，我们不必再依赖于外部变量，而是采用更新 showMessage 函数的方法来接受新消息。首先，在定义函数时要包括一个参数 Message，该参数是函数体中的变量。然后，当调用该函数时，将要显示的信息放在圆括号内，showMessage("It's full of stars!")。这个函数与代码清单 5-1 相比，增添了额外的文本，显示如下：

```
> The message is: It's full of stars!
```



代码清单 5-3　将信息传递给函数
(http://jsbin.com/xucemu/edit?js,console)

```
var showMessage;
showMessage = function (message){  //在定义函数时要包括一个参数 message
   console.log("The message is: " + message); //在函数内使用 message
};
showMessage("It's full of stars!");  //"It's full of stars!"传递给 Message，
                                     //函数体内的语句得以执行
```

在代码清单 5-3 的结尾处，调用 showMessage 函数时，将字符串"It's full of stars!"放在圆括号内。像以上示例中这样，在函数调用时放在圆括号内的值称作实参。程序将此参数赋值给变量 message，然后函数使用变量 message 来生成字符串并显示在控制台上，"The message is： It's full of stars!"

表 5-1 显示了具体步骤，教你如何定义一个带有形参的函数，然后使用不同的实参来调用函数。

表 5-1　定义和调用函数（传递数据给函数）的步骤

60

动　　作	代　　码	注　　释
声明变量	var showMessage;	命名一个变量以备在程序中使用
定义一个含参函数	function (message){ }	命名变量 message，以备在函数体内使用
使用参数	function (message){ console.log(message); }	参数是在函数体内可用的变量
将函数赋值给变量	showMessage = function (message){ console.log(message); };	将函数赋值给变量以备后续调用该函数
用实参来调用函数	showMessage("It's full of stars!");	将圆括号内的参数赋值给 message，执行函数体内的语句
使用不同的实参多次调用函数	showMessage("It's full of stars!"); showMessage("Yippee!"); showMessage("Cowabunga!");	每次调用该函数时，就赋予 message 不同的实参

如上所述，我们可以使用自己选择的任何文本内容来调用 showMessage 函数，将该文本赋值给 message，并作为完整消息的一部分显示在控制台上。

在代码清单 5-4 中，使用三个不同的参数调用 showMessage 函数，导致控制台上显示了三条不同的信息：

```
> The message is: It's full of stars!
> The message is: Hello to Jason Isaacs
> The message is: Hello to Jason Isaacs and Stephen Fry
```



代码清单 5-4　使用不同的参数调用同一个函数 (http://jsbin.com/zavavo/edit?js,console)

```
var showMessage;                          //声明一个变量
showMessage = function (message){         //用形参 message 定义函数，并将该函数赋值
                                          //给变量
    console.log("The message is: " + message);//在函数体内使用参数 message
};
showMessage("It's full of stars!");
showMessage("Hello to Jason Isaacs");
showMessage("Hello to Jason Isaacs and Stephen Fry");    //每次用不同的实
                                                         //参调用该函数
```

61

正是因为在定义函数的同时声明了形参的名称，函数 showMessage 不再依赖于程序的其他地方来声明变量，因此成功去除了函数和变量之间的绑定关系。

代码清单 5-5 定义了一个平方函数，包含一个形参 numberToSquare。函数将传递给它的数字作为实参进行平方运算。四次调用该函数得到以下输出：

```
> 10 * 10 = 100
> -2 * -2 = 4
> 1111 * 1111 = 1234321
> 0.5 * 0.5 = 0.25
```



代码清单 5-5　使用平方函数
(http://jsbin.com/vequpi/edit?js,console)

```
var square;
square = function (numberToSquare) {
    var result;
    result = numberToSquare * numberToSquare; //数值乘以自身得到平方数，*是
                                              //乘法运算符
    console.log(numberToSquare + " * " + numberToSquare + " = " + result);
};
square(10);
square(-2);
square(1111);
square(0.5);
```

形参与实参

我们早就知道，圆括号终究会派上用场！

在定义函数时，圆括号内的名称可以作为变量在函数体中使用，我们称之为形参，我们期望在调用函数时也使用该信息，如下所示：

```
var myExample;
myExample = function(parameter){...}
```

在调用函数时，将圆括号中的值传递给形参，并在函数体中运行，我们把调用函数时圆括号中的值称为实参，如下所示：

```
myExample(argument);
```

请读者不必过分担心这些术语，可能需要一段时间才能习惯使用这些术语。其实在练习创建和使用几个函数之后，就会对形参和实参有一个直观的认识，到那时，即便有时还会混淆这两个术语，那也无妨。 62

5.2.2　将多个实参传递给一个函数

可以根据需要，在定义一个函数时使用多个形参。具体做法就是在定义函数的圆括号中用逗号来分隔多个形参，如图 5-2 所示。

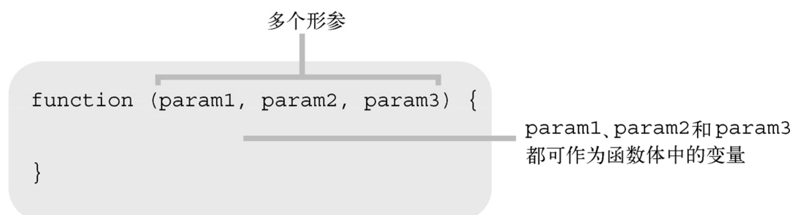


图 5-2　在定义一个函数时包括多个形参

假设我们需要一个函数来将两个数字相加。例如这两对数字分别是 30 和 23，以及 2.8 和-5，正确的输出应该是：

```
> The sum is 53
> The sum is -2.2
```

该如何编写代码？



代码清单 5-6　带有两个实参的函数
(http://jsbin.com/siyelu/edit?js,console)

```
var showSum;
showSum = function (number1, number2) { //在定义函数时使用两个形参：number1
                                        //和 number2
   var total = number1 + number2;
   console.log("The sum is " + total);
};
showSum(30, 23);   //使用两个实参调用该函数：30 和 23
showSum(2.8, -5);
```

当调用函数 showSum 时，程序会自动将两个实参赋值给定义函数时的两个形参：number1 和 number2。在代码清单 5-6 中，第一次调用函数 showSum 时，函数体就变成为：

```
var number1 = 30;
var number2 = 23;
var total = number1 + number2;
console.log("The sum is " + total);
```

我们在定义函数时可以随心所欲地使用很多的形参。但是随着形参数量的增加，其他人
63 （也包括你自己！）在使用函数时可能更容易出错，有可能会丢掉一个实参或者在调用函数时将实参的顺序搞错。为了解决这个问题，我们可以采用一个简便方法，就是将对象传递给函数。在定义函数时，只需要一个形参，函数体可以访问对象的任何一个属性。在后续第 7 章，读者将学习和使用这种带有对象的函数。

5.3　The Crypt——显示玩家信息

在本节中，我们把刚刚学到的 JavaScript 带参函数的知识应用到游戏 The Crypt 去。图 5-3 显示了本节的重点：通过使用带参函数来显示玩家信息。

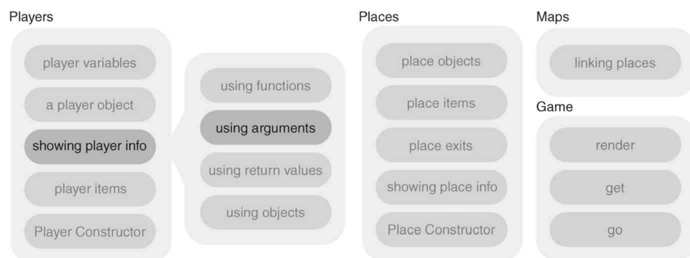


图 5-3　The Crypt 中的游戏元素

在第 4 章中，我们编写了一个函数 showPlayerInfo，在需要时可以调用该函数将玩家信

息显示在控制台上。但是该函数的存在依赖于程序中其他地方已经定义好的变量 player，现在我们通过设置参数，并向其直接传递所需信息的方法来更新函数 showPlayerInfo。

为了显示每一位玩家的多方面信息，可以为每位玩家的每条信息创建一个函数，如下所示：

```
showPlayerName("Kandra");
showPlayerHealth("Kandra", 50);
showPlayerPlace("Kandra", "The Dungeon of Doom");
```

以上每一个函数都有明确的内容。如果同时显示所有信息，可以将所有子函数打包成一个主函数，然后将其需要的所有信息传递给主函数，如下所示：

```
showPlayerInfo("Kandra", "The Dungeon of Doom", 50);
```

64

在后续几节内容中，我们定义了四个函数和若干玩家对象。前三个函数非常相似，请关注它们的异同点，并注意如何在定义函数时使用形参，在调用函数时使用实参。

5.3.1　显示玩家的姓名

第一个函数用于显示玩家的姓名，并无其他功能。以下代码清单显示如何定义函数 showPlayerName，并用两个不同的姓名参数来调用该函数，输出以下内容：

```
>Kandra
>Dax
```



代码清单 5-7　显示一个玩家的姓名
(http://jsbin.com/yubahi/edit?js,console)

```
var showPlayerName;
showPlayerName = function (playerName){     //在定义函数时包含一个形参 playerName
   console.log(playerName);                 //调用函数时给形参赋值并显示该值
};
showPlayerName("Kandra");
showPlayerName("Dax");
```

在真正的 The Crypt 程序中，可能不会使用"Kandra"和"Dax"来调用 showPlayerName 函数，而是使用变量来调用函数。特别值得一提的是，可以使用对象代表各个玩家。在下一个代码清单中，就是使用两个玩家对象 player1 和 player2 来更新程序。



代码清单 5-8　通过对象属性来显示玩家的姓名
(http://jsbin.com/juhewi/edit?js,console)

```
var player1;
var player2;
var showPlayerName;
showPlayerName = function (playerName) {
   console.log(playerName);
};
player1 = {
```

```
        name: "Kandra",
        place: "The Dungeon of Doom",
        health: 50
    };
    player2 = {
        name: "Dax",
        place: "The Old Library",
        health: 40
```

65

```
    };
    showPlayerName(player1.name);  //将玩家的姓名作为参数传递给函数 showPlayerName
    showPlayerName(player2.name);
```

以上是对玩家姓名的显示，下面将显示玩家的健康值。

5.3.2 显示玩家的健康值

在以下代码清单中，函数 showPlayerHealth 定义中包含了两个形参，playerName 和 playerHealth，输出如下：

```
>Kandra has health 50
>Dax has health 40
```



代码清单 5-9　显示玩家的健康值
(http://jsbin.com/nomija/edit?js,console)

```
var showPlayerHealth;
showPlayerHealth = function (playerName, playerHealth) { //在定义函数时
                                                        //包含两个形参：
                                               //playerName 和 playerHealth
    console.log(playerName + " has health " + playerHealth);
};
showPlayerHealth("Kandra", 50);         //为形参赋值来生成供显示的字符串
showPlayerHealth("Dax", 40);
```

在代码清单 5-9 中，调用函数时使用了字面值"Kandra"和 50。在下一个代码清单中，将每一个玩家的信息赋值给玩家对象的各个属性。事实上，我们在调用函数时，更倾向于使用这些属性，而不是使用这样的字面值。代码清单 5-10 就是使用玩家对象对程序进行了更新。



代码清单 5-10　通过对象属性来显示玩家的健康值
(http://jsbin.com/zufoxi/edit?js,console)

```
var player1;
var player2;
var showPlayerHealth;
showPlayerHealth = function (playerName, playerHealth) {
    console.log(playerName + " has health " + playerHealth);
};
player1 = {
```

```
    name: "Kandra",
    place: "The Dungeon of Doom",
    health: 50
};
player2 = {
    name: "Dax",
    place: "The Old Library",
    health: 40
};
showPlayerHealth(player1.name, player1.health);//将每一位玩家的姓名和健康值
                                               //属性作为参数传递给函数
                                               //showPlayerHealth
showPlayerHealth(player2.name, player2.health);
```

66

我们已经学会如何显示玩家的姓名和健康值，只剩下位置信息还未显示。

5.3.3　显示玩家的位置

在以下代码清单 5-11 中，对函数 showPlayerPlace 的定义也包含了两个参数：playerName 和 playerPlace，将输出以下信息：

```
>Kandra is in The Dungeon of Doom
>Dax is in The Old Library
```



代码清单 5-11　显示玩家的位置
(http://jsbin.com/yifahe/edit?js,console)

```
var showPlayerPlace;
showPlayerPlace = function (playerName, playerPlace) {
    console.log(playerName + " is in " + playerPlace);
};
showPlayerPlace("Kandra", "The Dungeon of Doom");
showPlayerPlace("Dax", "The Old Library");
```

以下代码清单 5-12 是代码清单 5-11 的更新版本，其中使用了对象属性，而非字面值。



代码清单 5-12　通过对象属性显示玩家的位置
(http://jsbin.com/mejuki/edit?js,console)

```
var player1;
var player2;
var showPlayerPlace;
showPlayerPlace = function (playerName, playerPlace) {
    console.log(playerName + " is in " + playerPlace);
};
player1 = {
    name: "Kandra",
    place: "The Dungeon of Doom",
    health: 50
```

```
};
player2 = {
    name: "Dax",
    place: "The Old Library",
    health: 40
};
showPlayerPlace(player1.name, player1.place);
showPlayerPlace(player2.name, player2.place);
```

67

我们看到，为了分别显示玩家的姓名、健康值和位置信息，总共需要使用三个函数。下面我们将把这三个函数合并为一个函数。

5.3.4 合并显示玩家的信息

函数 showPlayerInfo 使用了三个相互独立的函数：showPlayerName、showPlayerHealth 和 showPlayerPlace，并且增加了一些格式来显示每一位玩家的属性，输出信息如下所示：

```
>
>Kandra
> ----------------------------
>Kandra is in The Dungeon of Doom
>Kandra has health 50
> ----------------------------
>
```

以下代码清单为了聚焦显示新函数 showPlayerInfo，删减了以上三个独立的函数，不过在 JS Bin 上并未删减。



代码清单 5-13 显示一个玩家的信息 (http://jsbin.com/likafe/edit?js,console)

```
var showPlayerInfo;
showPlayerInfo = function (playerName, playerPlace, playerHealth) {
    console.log("");
    showPlayerName(playerName);
    console.log("----------------------------");
    showPlayerPlace(playerName, playerPlace);
    showPlayerHealth(playerName, playerHealth);
    console.log("----------------------------");
    console.log("");
};
showPlayerInfo("Kandra", "The Dungeon of Doom", 50);
showPlayerInfo("Dax", "The Old Library", 40);
```

在上述代码中，每次使用三个参数来调用函数，分别传递给函数 showPlayerName、showPlayerHealth 和 showPlayerPlace。以下代码清单 5-14 中，将所有的信息集合在一起，包括每一个变量声明和每一步变量赋值，并使用了玩家对象属性，例如 player1.name，而不是

字面值"Kandra"来调用函数 showPlayerInfo.。 68



代码清单 5-14 使用属性显示玩家的信息 (http://jsbin.com/loteti/edit?js,console)

```
var showPlayerName = function (playerName) {
   console.log(playerName);
};
var showPlayerHealth = function (playerName, playerHealth) {
   console.log(playerName + " has health " + playerHealth);
};
var showPlayerPlace = function (playerName, playerPlace) {
   console.log(playerName + " is in " + playerPlace);
};
var showPlayerInfo = function (playerName, playerPlace, playerHealth) {
   console.log("");
   showPlayerName(playerName);
   console.log("----------------------------");
   showPlayerPlace(playerName, playerPlace);
   showPlayerHealth(playerName, playerHealth);
   console.log("----------------------------");
   console.log("");
};
var player1 = {
   name: "Kandra",
   place: "The Dungeon of Doom",
   health: 50
};
var player2 = {
   name: "Dax",
   place: "The Old Library",
   health: 40
};
showPlayerInfo(player1.name, player1.place, player1.health);
showPlayerInfo(player2.name, player2.place, player2.health);
```

5.4 本章小结

- 定义一个带参函数，表明将在调用函数时，把数据传递给该函数。圆括号内的多个参数之间用逗号分隔开，如下所示：

```
function (param1, param2) { … }
```

- 在函数体内，把参数当作变量来使用。
- 用实参调用函数。在运行程序时，实参会自动赋值给形参，并在函数体内运行，例如：

```
myFunction(arg1, arg2)
```
69

第 6 章　返回值：从函数获取数据

本章内容包括：

■ 从函数返回信息

■ 关键字 return

■ 在控制台提示符下进行实验

在第 4 章，我们发现函数可以大大提高工作效率，因为只需一次性编写函数代码，就可以多次使用该函数。在第 5 章，我们学会了向函数传递信息来调用函数，使函数更加灵活好用：一个函数可以根据我们赋予它不同的参数而产生不同的输出结果。在本章，我们为函数赋予了另一个机会：将其运行结果返回到程序中。而且，我们还可以在控制台提示符下，通过直接调用函数来追踪该函数的返回值。

6.1　从函数返回数据

欲将函数为我们所用，非常有必要得到函数的工作结果，并可以随时使用该结果。在第 5 章代码清单 5-6 中，有一个函数 showSum，将两个数字之和显示在控制台上。如果有一个 add 函数，可以将数字相加并返回计算结果，那会更加理想。尽管代码清单 5-6 中的 showSum 确实可以在控制台上显示计算结果，但是如果使用 add 函数，不仅可以选择显示函数的返回值，而且可以在后续的计算中使用该返回值，还可以通过网络发送该返回值，或将

70 其保存到数据库中。

6.1.1　用返回值替代函数调用

到目前为止，我们编写的大多数函数都是按需执行代码, 然后将一些内容输出显示到控制台上。函数帮助我们把长的程序分解成可理解的若干片段，将这些函数赋值给命名好的变量使程序变得更加容易理解。以下在 The Crypt 中显示玩家信息的代码片段就是这样一个示例。

```
showPlayerName("Kandra");
showLine();
showPlayerPlace("Kandra", "The Dungeon of Doom");
showPlayerHealth("Kandra", 50);
showLine();
```

即使我们不知道这些函数体内部如何运行，这些函数名也很容易让人理解以上代码。

函数还可以返回计算结果、文本内容或数据库里的数据。我们可以将返回值赋值给变量，也可以将该返回值作为其他函数的参数使用。以下示例以粗体显示调用了四个函数

add、getPlayerPlace、findPlanetPosition 和 getMessage：

```
var sum = add(50, 23);
var placeInfo = getPlayerPlace("Kandra", "The Dungeon of Doom");
console.log(findPlanetPosition("Jupiter"));
console.log(getMessage());
```

以上每个函数返回一个值，返回值替代了函数调用。如果函数中以粗体表示返回值，则上述四个语句变成：

```
var sum = 73;
var placeInfo = "Kandra is in The Dungeon of Doom";
console.log("Jupiter: planet number 5");
console.log("I'm going on an adventure!");
```

图 6-1 显示了当调用函数 add 时会发生的情况。

欲从函数返回值，请使用关键字 return。

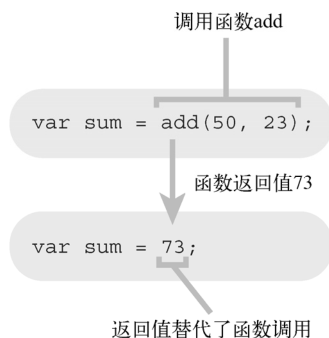


图 6-1　调用函数 add 得到返回值 73

71

6.1.2　关键字 return

使用关键字 return 可以返回函数的值。程序代码中关键字 return 之后的内容就是替代函数调用的返回值。

代码清单 6-1 定义了一个函数 getMessage，该函数包括一个 return 语句，该语句以关键字 return 作为开始标志：

```
return "I'm going on an adventure!";
```

因为在关键字 return 之后是字符串，所以该函数的返回值为字符串“I'm going on an adventure!”，程序将该字符串赋值给变量 response，并显示到控制台上：

```
> I'm going on an adventure!
```



代码清单 6-1　从函数获取返回值 (http://jsbin.com/yucate/edit?js,console)

```
var getMessage;
```

```
var response;
getMessage = function (){
    return "I'm going on an adventure!"; //使用关键字 return 从函数获取返回信息
};
response = getMessage();//调用函数 getMessage，并将返回值赋值给变量 response
console.log(response);
```

在以上代码中，函数 getMessage 总是返回相同的值。在编程实践中，我们通常是在调用函数时将参数传递给函数，以此得到函数的返回值。

6.1.3 使用参数来确定返回值

在第 5 章，我们学习了如何在函数定义中使用形参，然后在函数调用中使用实参将信息传递到函数中。这样，我们就可以在函数体中使用该信息来确定函数返回的值，因此使用不同的参数反复调用函数，就会产生不同的返回值。

下面的代码清单显示 getHelloTo 函数返回一个包含参数 name 的字符串，程序将返回值赋给变量并将其记录到控制台上：

```
> Hello to Kandra
```



代码清单 6-2 使用参数来确定返回值
(http://jsbin.com/nijijo/edit?js,console)

72

```
var getHelloTo;
var fullMessage;
getHelloTo = function (name) {      //在函数定义中使用形参
    return "Hello to " + name;
};
fullMessage = getHelloTo("Kandra");      //通过对形参赋值来构建返回值
console.log(fullMessage);
```

在代码清单 6-2 中，将返回值赋值给变量 fullMessage，并将其记录到控制台上。其实使用该变量是多余的，完全可以直接记录返回值，如以下代码清单 6-3 所示。在代码清单 6-3 中，两次调用函数 getHelloTo，得到以下输出：

```
> Hello to Kandra
> Hello to Dax
```



代码清单 6-3 使用返回值作实参
(http://jsbin.com/yapic/edit?js,console)

```
var getHelloTo;
getHelloTo = function (name) {
    return "Hello to " + name;
};
console.log(getHelloTo("Kandra")); //调用函数 getHelloTo 并且使用该函数的返
                                   //回值作为 console.log 的实参
console.log(getHelloTo("Dax"));
```

函数可以返回任何类型的值：字符串、数字、对象甚至其他函数。下面我们看一个返回数字的例子。

图 6-2 所示为调用函数 add(50，23)。该图显示如何将实参传递给函数 add，再以此计算返回值。

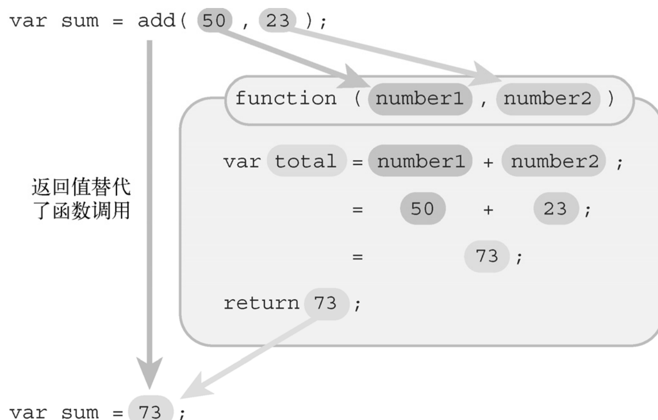


图 6-2　函数 add 计算返回值

73

下一个代码清单展示了函数 add 的运行情况，请特别注意其中的关键字 return。



代码清单 6-4　返回两数相加之和
(http://jsbin.com/haqapu/edit?js,console)

```
var add;
add = function (number1, number2) {
   var total = number1 + number2;   //将计算结果赋值给变量 total
   return total;         //使用关键字 return 将计算结果返回到调用函数的地方
};
var sum = add(50, 23);  //调用函数 add，并将函数的返回值赋值给变量 sum
console.log(sum);
```

在以上代码中，add(50, 23)调用了函数 add，将 50 赋值给 number1，将 23 赋值给 number2，number1 的值与 number2 的值相加，两者相加之和赋值给变量 total。函数体内最后一行的关键字 return，用 total 值替代了函数调用，因此，以下

```
var sum = add(50, 23);
```

就变成：

```
var sum = 73;
```

因为 add(50, 23)的返回值为 73。

请注意，赋值运算符“＝”将其右侧的值赋值给其左侧的变量。如果函数调用在“＝”的右侧，则将该函数的返回值赋值给其左侧的变量。

在以上代码清单 6-4 中，函数 add 中的变量 total 并不是必需的，因为它被赋值给一个

变量并立即返回。正如代码清单 6-5 所示，函数 totalCost 可以将计算结果直接返回。基于水管工的上门服务费和水管工的每小时收费率，totalCost 可以算出其工作若干小时后的总费用。例如，水管工的上门服务费是$30，工作每小时收费$40，那么他工作三小时的总费用是多少？ totalCost(30，40，3)应该返回结果$150。



代码清单 6-5　含三个参数的函数
(http://jsbin.com/jedigi/edit?js,console)

```
var totalCost;
totalCost = function (callOutCharge, costPerHour, numberOfHours) { //计算总费用并返回值
    return callOutCharge + costPerHour * numberOfHours;
};
console.log("$" + totalCost(30, 40, 3)); //函数返回值替代了调用函数 totalCost
```

74

在以上代码中，当调用函数 totalCost 时，先对关键字 return 右边进行计算，然后返回值，该计算遵循通常的算术规则，首先执行乘法，然后执行加法：$30 + 40\times3 = 30 + 120 = 150$。

6.2　在控制台提示符下进行实验

程序员通常不会在程序成型阶段使用控制台。也就是说，在程序发布后和在线使用中，程序员通常不再使用控制台。他们通常在开发阶段，即程序设计和编写时使用控制台。控制台为程序员提供了一种方便的方法来记录程序在运行中的值和出现的错误，并可以跟踪调查程序在运行时的各种状态。这种交互性，即获得即时反馈的机会，使控制台对于编程学习非常有帮助——尤其是通过实验来学习编程。下面我们就去 JS Bin 控制台探险和发现上述函数功能。

6.2.1　调用函数

以下代码清单 6-6 包括前几个代码清单的四个函数，在运行程序时，不会在控制台上产生任何输出，不包括对 console.log 的任何调用。



代码清单 6-6　几个有返回值的函数
(http://jsbin.com/lijufo/edit?js,console)

```
var getMessage;
var getHelloTo;
var add;
var totalCost;
getMessage = function () {
    return "I'm going on an adventure!";
};
getHelloTo = function (name) {
    return "Hello to " + name;
};
```

```
add = function (number1, number2) {
   return number1 + number2;
};
totalCost = function (callOutCharge, costPerHour, numberOfHours) {
   return callOutCharge + costPerHour * numberOfHours;
};
```

运行以上程序，将四个函数分别赋值给在程序顶部声明过的四个变量，然后，我们就可以在控制台提示符下访问这些变量，可以调用那些已经赋值给变量的函数，可以在控制台上看到这些函数的返回值。

单击JS Bin上的链接，并运行该程序。在控制台提示符后面输入：

```
> getMessage()
```
75

然后输入〈Enter〉键，函数getMessage得以运行，并在控制台上显示其返回值:

```
"I'm going on an adventure!"
```

在控制台提示符后面按键盘上的向上箭头，就会打开刚刚输入的最后一行内容。可以使用控制台上的向上和向下箭头来找到先前输入的条目，按〈Enter〉键可以重新提交该命令。以上代码中，函数getMessage 总是返回相同的字符串。如果查看该函数的函数体，就可以看到它返回的确实是字符串形式的文字，是一个硬编码值，如下所示：

```
return "I'm going on an adventure!";
```

以上程序中的另外一个函数getHelloTo在其定义中包含了一个形参name。形参name的意义在于允许该函数的返回值产生变化。当每次调用该函数时传递给该函数的实参改变了，那么该函数的返回值会有相应的改变，如下所示：

```
getHelloTo = function (name) {
   return "Hello to " + name;
};
```

可以在控制台上尝试每次用不同的参数来调用该函数, 如下所示:

```
> getHelloTo("Jason")
"Hello to Jason"
> getHelloTo("Rosemary")
"Hello to Rosemary"
```

可以看到，参数的改变会引起函数返回值的改变。

6.2.2　声明新的变量

控制台不仅允许用户访问已声明的变量，而且允许用户声明新的变量并为其赋值。

紧接着上一节内容，在控制台上输入：

```
> var friend
```

并按〈Enter〉键，控制台上依旧是前一个条目的值，因为我们尚未对变量 friend 赋值，控制台上会显示：undefined（未定义）。

给变量 friend 赋值，输入：

```
> friend ="Amber"
```

并按〈Enter〉键。控制台上会显示新值：

76

```
"Amber"
```

程序会使用新的变量friend作为函数getHelloTo的参数并在控制台上显示：

```
> getHelloTo（friend）
"Hello to Amber"
```

再来试一试函数 add 和 totalCost。向它们传递不同的参数，并查看在控制台上显示的返回值，如下所示：

```
> add（30，12）
42
> totalCost（10，20，3）
70
```

控制台不仅可以记录来自程序的消息，用户还可以通过控制台深入挖掘程序，并使用控制台来查看程序的运行是否如预期所料。不需要任何批准或许可，读者就可以直接测试本书中的所有代码清单。

下面我们使用刚刚掌握的返回值知识来进一步改进游戏 The Crypt 中显示玩家信息的方式。

6.3 The Crypt——构建玩家信息字符串

图 6-3 显示了本节的重点内容：通过使用返回值显示玩家的信息。

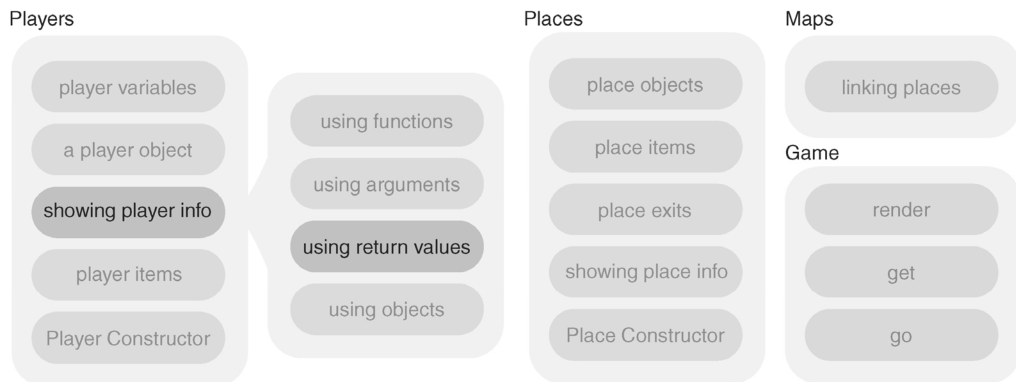


图 6-3　The Crypt 中的游戏元素

在第 5 章，我们将显示玩家信息的任务分配给若干个函数，每个函数都有一个特定的任务，如下所示：

```
showPlayerName（"Kandra"）；
showPlayerPlace（"Kandra"，"The Dungeon of Doom"）；
showPlayerHealth（"Kandra"，50）；
```

以上每个函数都将一串信息记录到控制台上，但是如果我们并不想在控制台上显示以上信息，该如何是好？例如，我们希望在电子邮件中发送以上信息，我们希望将其作 77
为对 Web 请求的响应，或者我们希望将其附加到网页上的某个现有元素上。如果一个函数不仅构建了信息字符串，并且可以供程序根据需要使用其返回值，那么该函数就会变得更加灵活好用。

6.3.1　为玩家的姓名、健康值和位置构建字符串

将函数所需要的信息作为参数传递给这些函数，这些函数就会返回一个包含该信息的字符串，如下所示：

```
getPlayerName("Kandra");        //--> "Kandra"
getPlayerHealth("Kandra", 50);     //--> "Kadra has health 50"
getPlayerPlace("Kandra", "The Dungeon");   //-->"Kandra is in The Dungeon"
```

首先分析getPlayerName函数。目前该函数只是返回一个传递给它的姓名，貌似这是浪费时间。但是，定义这样一个函数，能够让我们以同样的方式访问所有的玩家信息，并且获取玩家的姓名信息与获取玩家的健康值和位置信息三者大同小异。定义这样一个函数使得以后的代码更新变得更容易，例如更改姓名的显示方式。以下代码清单显示了getPlayerName的函数定义和一次示范性调用该函数，输出结果如下所示：

```
> Kandra
```



代码清单 6-7　为玩家的姓名构建字符串 (http://jsbin.com/hijeli/edit?js,console)

```
var getPlayerName;
getPlayerName = function (playerName) {
   return playerName;
};
console.log(getPlayerName("Kandra"));
```

我们已经学会使用 getPlayerName 为玩家的姓名构建字符串，再来学习下面两个函数 getPlayerHealth 和 getPlayerPlace 就变得非常容易，因为这三个函数非常相似，都是根据传递来的信息构建一个简单的字符串。下一个代码清单包括对 getPlayerHealth 和 getPlayerPlace 两个函数的定义以及一些使用示例，会在控制台上输出以下信息：

```
> Kandra has health 50
> Kandra is in The Dungeon of Doom
```



代码清单 6-8　为玩家的健康值和位置构建字符串
(http://jsbin.com/pemore/edit?js,console)

78

```
var getPlayerHealth;
var getPlayerPlace;
getPlayerHealth = function (playerName, playerHealth) {
   return playerName + " has health " + playerHealth;
};
getPlayerPlace = function (playerName, playerPlace) {
   return playerName + " is in " + playerPlace;
};
console.log(getPlayerHealth("Kandra", 50));
console.log(getPlayerPlace("Kandra", "The Dungeon of Doom"));
```

如何使用一个主函数 getPlayerInfo 来调用以上三个函数，向我们显示所需要的字符串？下一节内容将展示如何将这几个函数合并为一个主函数。

6.3.2　用一个函数显示玩家的信息——把几个函数集合在一起

现在要把几个函数集合在一起来构建一个主函数，生成一个关于玩家的信息字符串，输出如下所示：

```
>
> Kandra
> ********************
> Kandra is in The Dungeon of Doom
> Kandra has health 50
> ********************
>
```

为了构造一行字符来分隔玩家的各种信息，可以使用函数 getBorder，该函数的返回值是一行星号，如上所示。下面的代码清单显示了函数 getPlayerInfo 的定义，其中包括对某些其他函数的调用，这些函数在 JS Bin 中是可以看到的。



代码清单 6-9　创建一个显示玩家信息的字符串
(http://jsbin.com/javuxe/edit?js,console)

```
var getPlayerInfo;
getPlayerInfo = function (playerName, playerPlace, playerHealth) {
   var playerInfo;
   playerInfo = "\n" + getPlayerName(playerName); //通过使用其他函数来构
                                                  //建字符串的一部分内容
   playerInfo += "\n" + getBorder();//通过使用+=来追加一个字符串到现有字符串
   playerInfo += "\n" + getPlayerPlace(
      playerName, playerPlace);
   playerInfo += "\n" + getPlayerHealth(
      playerName, playerHealth);
```

```
    playerInfo += "\n" + getBorder();
    playerInfo += "\n";
    return playerInfo;   //返回完整的字符串
};
console.log(getPlayerInfo("Kandra", "The Dungeon of Doom", 50));
```

在以上代码中，通过把调用函数返回的字符串一步步地追加给变量 playerInfo，函数 getPlayerInfo 逐步构建完整的玩家信息字符串。在每个步骤中，附加的额外字符串\n 是一个换行字符，换行符后面的文本将出现在控制台的一个新行上。使用+=运算符可以将一个字符 79 串追加到现有字符串的末尾。

本章的最后一个例子是将所有代码集合到代码清单 6-10 中，并且用两个玩家对象来测试函数 getPlayerInfo。在该代码清单中，函数 getBorder 使用了不同的分隔符来分隔信息，输出两个玩家的信息如下所示：

```
>
> Kandra
> ================================
> Kandra is in The Dungeon of Doom
> Kandra has health 50
> ================================
>
>
> Dax
> ================================
> Dax is in The Old Library
> Dax has health 40
> ================================
>
```



代码清单 6-10　使用对象来显示玩家的信息
(http://jsbin.com/puteki/edit?js,console)

```
var getPlayerName = function (playerName) {   //定义函数来返回玩家具体信息
                                              //的字符串
    return playerName;
};
var getPlayerHealth = function (
    playerName, playerHealth) {
        return playerName + " has health " + playerHealth;
};
var getPlayerPlace = function (
    playerName, playerPlace) {
        return playerName + " is in " + playerPlace;
};
var getBorder = function () { //定义函数来返回一行字符串作为分隔符
    return "================================";
```

```
};
var getPlayerInfo = function (
    playerName, playerPlace, playerHealth) {  //定义函数来调用先前的几个函
                                              //数，构建玩家信息字符串
    var playerInfo;
    playerInfo = "\n" + getPlayerName(playerName);
    playerInfo += "\n" + getBorder();
    playerInfo += "\n" + getPlayerPlace(playerName, playerPlace);
80  playerInfo += "\n" + getPlayerHealth(playerName, playerHealth);
    playerInfo += "\n" + getBorder();
    playerInfo += "\n";
    return playerInfo;
};
var player1 = {          //创建两个玩家对象
    name: "Kandra",
    place: "The Dungeon of Doom",
    health: 50
};
var player2 = {
    name: "Dax",
    place: "The Old Library",
    health: 40
};
console.log(       //用两个玩家测试该函数，将返回的字符串显示在控制台上
    getPlayerInfo(
        player1.name, player1.place, player1.health));
console.log(
    getPlayerInfo(
        player2.name, player2.place, player2.health));
```

现在，我们不仅能在控制台上显示输出的函数，而且可以进一步得到玩家信息字符串的返回值，随后还可以选择用这些字符串来做些事情。

将玩家的各种信息分别传递给函数也是一件麻烦事。因为不同的函数需要不同的信息，所以我们必须确保按正确的顺序放置参数。如果能够将整个玩家对象作为参数传递给函数，再让函数访问它所需要的属性，那么这件麻烦事将会变得轻松许多。在第 7 章中，我们将会看到，把 JavaScript 对象既用作参数又用作返回值会带给我们许多便利！

6.4 本章小结

■ 使用关键字 return 返回函数的信息：

```
return "Pentaquark!";
return 42;
return true;
```

- 将函数调用得到的值赋值给变量或用作参数，用函数的返回值替代函数调用：

```
var particle = getDiscovery();
console.log(getPlayerPlace("Kandra"));
```
81

- 定义包含形参的函数，使用实参向函数传递信息，并以此确定函数的返回值：

```
var sum = add（28，14）;
```

- 使用控制台来浏览和测试程序。
- 在控制台提示符下调用函数，并查看函数的返回值。
- 在控制台上声明新的变量，并为其赋值，然后用作参数传递给函数。 82

第 7 章　对象参数：将对象传递给函数

本章内容包括：

- 使用对象作为参数
- 在函数内部访问对象的属性
- 在函数内部添加新的对象属性
- 从函数返回对象
- 将函数设置为对象的属性

本书第 4～7 章一直在讲述函数的相关内容，本章是这四章中的最后一章。在前面几章中，我们已经学会如何根据需要使用函数执行代码，并且学会了如何使用形参、实参和关键字 return 向函数传递和返回信息。我们还学习了如何使用对象将各个值集合到一起，这些值称为对象的各个属性。下面我们将把函数和对象结合到一起，二者的结合会像涡轮增压一样大大提高程序的效率和可读性。

大家还记得第 3 章中提到的急救包吗？我们把急救包看作一个整体对象来传递和收纳。这样的打包或分块处理方式，也就是将个体集合视为单一整体的处理方式，是人类在语言和记忆中处理复杂信息时采用的一种重要方法。当我们真正需要内部物品时，才会考虑急救包内的各个元素：防腐剂、膏药、绷带等。

在本章中，我们将使用与急救包相同的打包方式，把对象作为一个整体参数向函数传递并返回信息。

7.1　使用对象作为参数

在编程时，如果能够将对象作为一个整体传递给函数，将会非常有用，尤其是该对象具有许多属性，而函数又需要访问这些属性时，使用对象作为函数的参数就显得尤为方便了。我们只需定义一个含参函数，在调用函数时就不再需要从前那个很长的实参列表了。与 show PlayerInfo（player1.name，player1.location，player1.health）相比，showPlayerInfo（player1）显然更加简单整洁，更加容易理解，也更加不易出错。

欲将相同的信息传递给某一函数，我们完全不必将各信息作为单独的值进行传递，而是采用将各信息打包为单一对象的方式进行传递（图 7-1）。



图 7-1　与多个参数相比，一个对象作为一个参数显然更加简洁

7.1.1　访问对象参数的属性

出于对太空冒险、Rosetta 和 Philae 航天器充满向往和好奇心，我们决定编写一个应用程序来显示有关太阳系的信息。这款应用程序的功能之一就是显示关于各行星的信息。

以下代码清单 7-1 显示了一个带有参数 planet 的函数 getPlanetInfo。当使用对象 planet 作为参数调用该函数时，函数体返回一个字符串，该字符串是基于对象 planet 的某些属性构建出来的，输出以下内容：

```
>Jupiter: planet number 5
```



代码清单 7-1　将对象作为参数传递给函数 (http://jsbin.com/tafopo/edit?js,console)

```
var planet1;
var getPlanetInfo;
planet1 = {
    name: "Jupiter",                                              84
    position: 5,
    type: "Gas Giant",
    radius: 69911,
    sizeRank: 1
};
getPlanetInfo = function (planet){ //包含一个形参 planet，以供实参对象
                                   //planet 传入函数
    return planet.name+ ": planet number " + planet.position;//访问对象
                                                             //planet 的属性
};
console.log(getPlanetInfo(planet1));     //调用函数 getPlanetInfo，将一个
                                         //行星作为实参传入
```

在以上代码中，将对象作为单个参数传递给函数使程序变得干净整洁。我们再也没有必要耗费精力去确保按照正确的顺序添加所有必需的参数，因为一个对象参数就解决了问题。

将对象传递给函数后，JavaScript 会自动将其分配给函数定义中所包含的参数。

我们可以通过参数访问对象的属性，如图 7-2 所示。该函数通过参数完全控制了这个对象，甚至可以为该对象添加新属性。

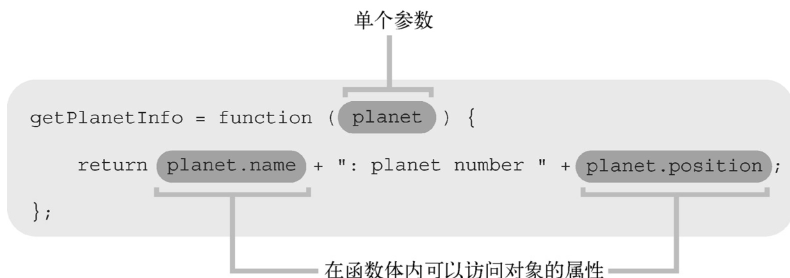


图 7-2　在函数体内可以访问对象的属性

7.1.2　给对象参数添加属性

当我们将对象作为参数传递给函数时，函数体中的代码有权访问对象的属性，可以读取、更改和删除对象的属性，并可以给对象添加新的属性。

代码清单 7-2 显示了两个带有参数 planet 的函数。当我们将一个行星对象（包含行星名称和半径）传递到函数 calculateSizes 时，该函数向该行星对象添加了两个新的属性：area 和

85 volume（面积和体积），如图 7-3 所示。

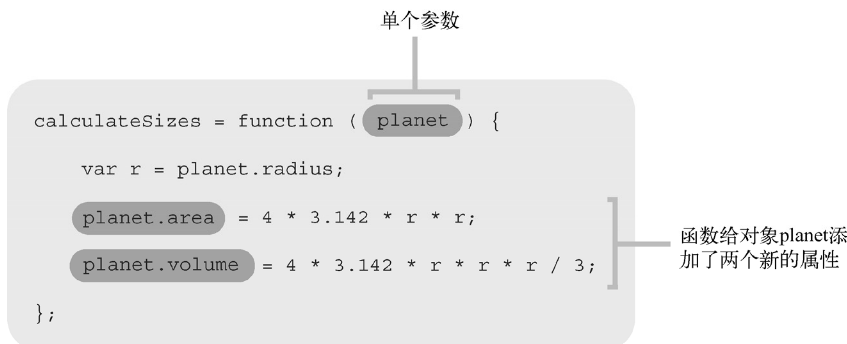


图 7-3　函数 calculateSizes 给对象 planet 添加了新属性

函数 displaySizes 使用两个新属性将该行星的有关信息显示在控制台上：

```
>Jupiter
>surface area = 61426702271.128 square km
>volume = 1431467394158943.2 cubic km
```



代码清单 7-2　给对象添加属性的函数
(http://jsbin.com/qevodu/edit?js,console)

```
var planet1 = { name: "Jupiter", radius: 69911 };
var calculateSizes = function (planet){ //在函数 calculateSizes 定义时，只
                                        //有一个参数 planet
    var r = planet.radius;
    planet.area= 4 * 3.142 * r * r;     //在函数调用时，给对象 planet 增加
                                        //了一个面积属性 area
    planet.volume= 4 * 3.142 * r * r * r / 3; //增加了一个体积属性 volume
```

```
};
var displaySizes = function (planet){    //在函数 displaySizes 定义时，只有
                                         //一个参数 planet
    console.log(planet.name);
    console.log("surface area = " + planet.area + " squarekm");
    console.log("volume = " + planet.volume + " cubickm");
};
calculateSizes(planet1);       //将对象 planet1 作为参数传递给两个函数
displaySizes(planet1);
```

在以上代码中，创建了对象 planet，并将其赋值给变量 planet1：

```
planet1 = { name: "Jupiter", radius: 69911 };
```

86

当调用函数 calculateSizes 时，将 planet1 作为实参传递给该函数：

```
calculateSizes(planet1);
```

JavaScript 将 planet1 赋值给形参 planet 以供该函数使用。该函数使用参数 planet 为对象添加两个新属性 planet. area 和 planet. volume。通过调用 displaySizes 函数，将该对象的两个新属性的信息显示出来：

```
displaySizes(planet1);
```

不但可以将对象传递给函数，而且可以从函数返回对象。

7.2　从函数返回对象

在第 6 章，我们学习了如何把对象作为参数传递给函数，这是一种将信息移动到所需位置的好方法。同理，使用对象作为函数的返回值也同样高效可行。函数可以对传递给它们的对象进行处理，然后将对象返回，或者函数也可以返回一个在函数体内创建的新对象。

本节探讨了两个例子：在第一个例子中，使用很多参数来构建一个新的行星对象，在第二个例子中，使用两个对象参数来创建二维空间中的一个点。

7.2.1　构建行星——对象创建函数

重返前文中提过的太阳系应用程序，现在我们决定简化对行星的创建程序。我们打算编写一个函数来传递关键信息，并返回一个有相应属性的行星对象 planet。我们使用函数 buildPlanet 创建行星对象，如下所示：

```
planet1 = buildPlanet（"Jupiter"，5，"Gas Giant"，69911，1）;
```

图 7-4 显示 JavaScript 在调用函数 buildPlanet 时如何将实参赋值给形参。我们就这样使用参数来创建一个新对象。

代码清单 7-3 定义了函数 buildPlanet，另外还有一个函数 getPlanetInfo 用于获取显示行星的信息，显示信息如下所示：

```
JUPITER: planet 5
NEPTUNE: planet 8
```

请注意星球名称采用大写字母。函数 getPlanetInfo 使用 JavaScript 的内置函数 toUpperCase，将字符串转换为大写字母。上述 toUpperCase 和其他一些 JavaScript 的内置函

87 数将在 7.3 节中讨论。

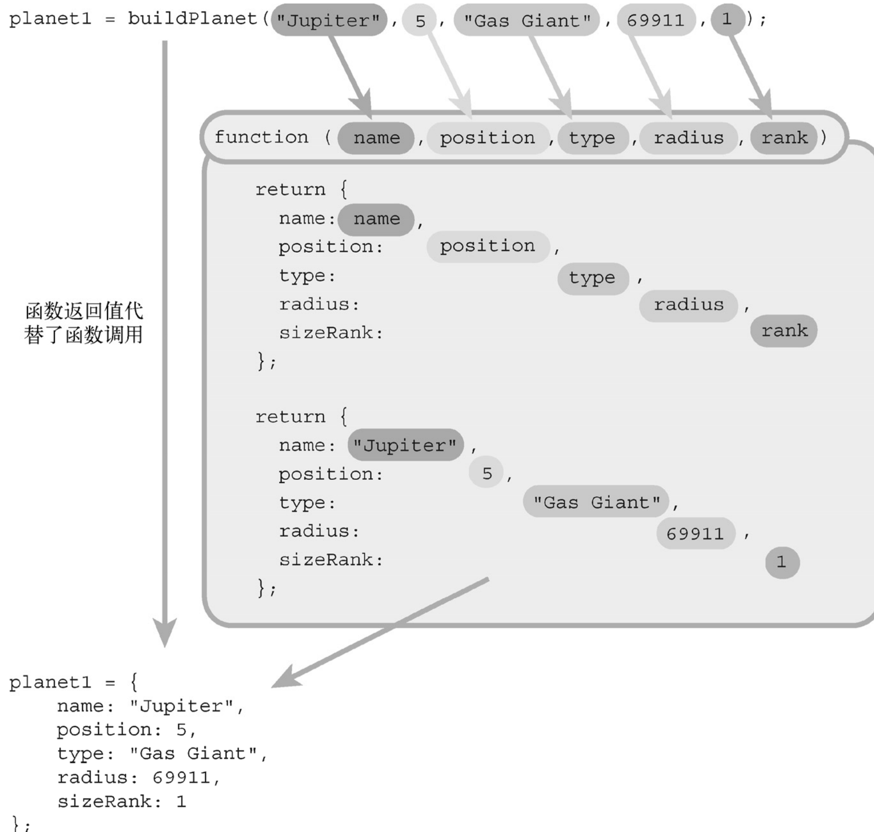


图 7-4　函数 buildPlanet 使用形参来创建一个对象，当调用函数时 JavaScript 将实参赋值给形参



代码清单 7-3　用函数来创建行星
(http://jsbin.com/coyeta/edit?js,console)

```
var buildPlanet;
var getPlanetInfo;
var planet1;
var planet2;
buildPlanet = function (name, position, type, radius, rank){
    return {          //使用花括号和从函数立即返回值来创建一个对象
        name: name, //创建一个属性 name，并将参数 name 的值赋给属性 name
        position: position,
        type: type,
        radius: radius,
```

88

```
        sizeRank: rank
    };
};
getPlanetInfo = function (planet) {
    return planet.name.toUpperCase() + ": planet " + planet.position;
};
planet1 = buildPlanet("Jupiter", 5, "Gas Giant", 69911, 1);
planet2 = buildPlanet("Neptune", 8, "Ice Giant", 24622, 4);
console.log(getPlanetInfo(planet1));
console.log(getPlanetInfo(planet2));
```

由函数 buildPlanet 创建的对象中的键-值对，看起来有些奇怪：

name：name，position：position，等等。

对于每个键-值对，键位于冒号的左侧，值位于冒号的右侧。我们使用形参作为值，因此对于以下函数调用：

```
planet1 = buildPlanet（“Jupiter”，5，“Gas Giant”，69911，1）；
```

这个对象创建代码就变为：

```
name: "Jupiter",
position: 5,
type: "Gas Giant",
radius: 69911,
sizeRank: 1
```

7.2.2　二维空间的点

为了把这些行星以形象的方式展现在太阳系中，我们需要使用二维坐标。每一个点的坐标都有 x 和 y 两个值，这看上去就像每一个位置点是一个对象，而该对象拥有 x 和 y 两个属性，如下所示：

```
point1 = { x : 3 , y : 4 };
point2 = { x : 0 , y : -2 };
```

我们可以用圆点运算符来访问每一个值，如 point1.x, point2.y 等。

作为首次实验，我们可以编写一个程序来移动一个位置点，使该点在 x 方向和 y 方向上各移动一定的距离。因为位置的变化包括 x 和 y 两个要素，因此同样可以使用对象来表示。例如，可以使用对象{ x : 4,y : -2 }来表示向右移动 4，向下移动 2。

以下代码清单 7-4 中的函数 move 包含两个参数，一个初始位置对象和一个位置移动对象。如果从第一个位置点出发，移动指定的数值，该点就变成了一个代表最终位置的新位置点。 89

该程序使用了一个函数 showPoint，输出结果如下：

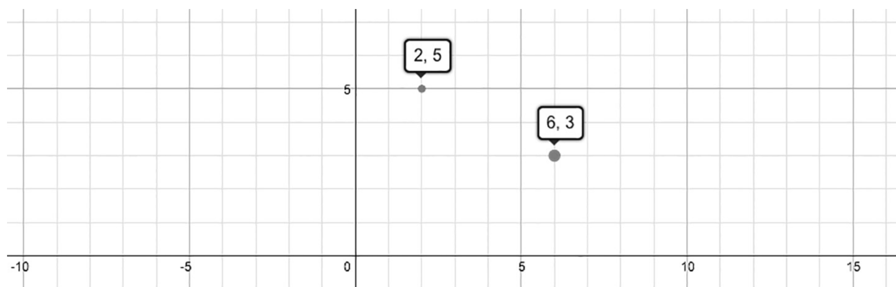
```
( 2 , 5 )
Move 4 across and 2 down
( 6 , 3 )
```



代码清单 7-4　在二维空间移动一个位置点
(http://jsbin.com/baxuvi/edit?js,console)

```
var point1;
var point2;
var move;
var showPoint;
move = function (point, change) {
    return {                    //使用花括号创建一个对象，并立即返回值
        x: point.x + change.x, //通过将位置变化值应用到初始位置点上来设定新的坐标
        y: point.y + change.y
    };
};
showPoint = function (point) {
    console.log("( " + point.x + " , " + point.y + " )");
};
point1 = { x : 2, y : 5 };
point2 = move(point1, { x : 4, y : -2 }); //把初始位置点作为一个变量，把位
                                          //置移动值作为对象字面量，传递给函数 move
showPoint(point1);
console.log("Move 4 across and 2 down");
showPoint(point2);
```

在以上代码清单 7-4 中，传递给函数 move 的第二个参数是一个对象字面量{ x : 4, y : -2}。其实我们也可以先将其赋值给一个变量，但是在程序中我们仅使用它一次，因此使用变量毫无意义，使用对象字面量就足够了。这两个位置点的坐标如图 7-5 所示。



90 图 7-5　初始位置点和（向右移动 4，向下移动 2 之后的）新位置点

从上述代码清单 7-4 可见，对象可以作为参数传递给函数，也可以作为返回值从函数返回。像这样，函数可以使用对象作为参数，也可以把对象作为函数的返回值，这就是使用对象的函数。在下一节内容中，我们将学习如何将函数设定为对象的属性。

7.3　方法——设置函数作为对象的属性

在 JavaScript 中，我们可以将数字、字符串和对象作为值，同样也可以将函数作为值。这意味着我们不仅可以将函数作为参数进行传递，可以将函数作为其他函数的返回值，还可

以将函数设置为对象的属性。在本节中，将继续探讨使用对象的函数，读者将看到一个将函数设置作为对象属性的示例。

7.3.1　命名空间——将相关函数组织到一起

下面我们编写一些函数对控制台上的文本进行格式化。显示文本是交互式控制台应用的一个重要内容，因此格式化函数的数量也会越来越多。首先，学习两个非常简单的函数：blank，用于返回一个空字符串；newLine，用于返回一个换行符。

当几个函数都与同一个工作相关时，最好将它们收集到一起。为此，我们可以将每个函数设置为某个对象的属性，该对象称为 spacer：

```
var spacer = {};
```

一旦我们拥有了一个对象，就可以将函数设置为该对象的属性。首先，用下面这个函数来返回一个空字符串：

```
spacer.blank = function () {
   return "";
};
```

图 7-6 所示为如何创建该函数，并将其赋值给对象 spacer 的属性 blank。

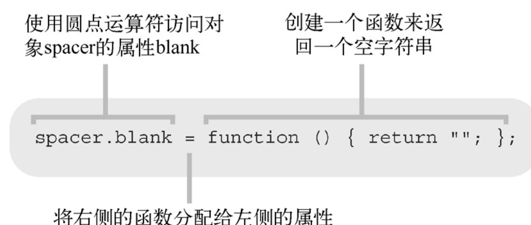


图 7-6　创建一个函数，并将其赋值给一个对象的属性 91

有时，我们需要在字符串中包含换行符，以便分隔开多行内容，"\n"为换行符。以下函数的返回值就是这样一个换行符：

```
spacer.newLine = function () {
   return "\n";
};
```

现在我们已经将两个函数设置为对象 spacer 的属性。像这样，使用一个对象将几个函数收集在一起，称之为命名空间（namespace）。函数 newLine 和 blank 都属于命名空间 spacer。我们依然可以通过添加圆括号的方法来调用这些函数：

```
console.log(spacer.blank());
console.log("Line1" + spacer.newLine() + "Line2");
console.log(spacer.blank());
```

该代码段会生成以下输出：

```
>
>Line1
>Line2
>
```

不必如上例所示，把各函数逐一添加到一个命名空间中。我们可以使用对象字面量语句，也可以使用花括号包含多个用逗号分隔的键-值对来设置带有多个属性的命名空间，如下所示：

```
spacer = {
    blank: function () {
        return "";
    },
    newLine: function () {
        return "\n";
    }
};
```

把函数设置为对象的属性称为方法（methods）。目前，对象 spacer 拥有两个方法：blank 和 newLine。我们将在第 7.3.4 和 7.3.5 节中为其添加更多方法，并在 JS Bin 上进行试验。

JavaScript 包括许多有用的对象和方法。在扩展对象 spacer 之前，我们先来研究 Math 方法和 String 方法。

7.3.2 Math 方法

Math 是一个内置于 JavaScript 中的命名空间，提供所有与数学计算有关的属性和函数。
92 代码清单 7-5 显示运行中的 Math.min 和 Math.max 方法，它们分别返回两个数字中较小的数字和两个数字中较大的数字， 程序产生以下输出：

```
3 is smaller than 12
-10 is smaller than 3
```



代码清单 7-5　使用 Math.min 和 Math.max (http://jsbin.com/moyoti/edit?js,console)

```
var showSmaller = function (num1, num2) {
    var smaller = Math.min(num1, num2);     //将两个数字中较小的数字赋值给 smaller
    var larger = Math.max(num1, num2);      //将两个数字中较大的数字赋值给 larger
    console.log(smaller + " is smaller than " + larger);
};
showSmaller(12, 3);
showSmaller(-10, 3);
```

Math.min 和 Math.max 一起使用有助于确保一个值能够在指定范围内。例如一个 lineLength 变量必须在 0 到 40 之间，包含 0 和 40。欲强制 lineLength 大于或等于 0，就可以使用：

```
lineLength = Math.max（0，lineLength）;
```

如果 lineLength 大于 0，它是两者当中最大的，其值不会改变。但是，如果 lineLength 小于 0，则 0 是两者当中最大的，lineLength 将被赋值为 0。

类似地，可以强制 lineLength 小于或等于 40：

```
lineLength = Math.min（40，lineLength）;
```

以下代码清单 7-6 显示如何实现上述对数值的限制。函数 line 的返回值是指定长度的分隔线，该指定长度必须介于 0 和 40 之间，当指定长度为 30，40 和 50 时，会产生以下输出：

```
> ==============================
> ========================================
> ========================================
```

请注意，最后两行线的长度都是 40。尽管最后一行线的指定长度是 50，但是函数 line 将长度限制为 40。



代码清单 7-6　使用 Math.min 和 Math.max 来限制参数 (http://jsbin.com/qiziyo/edit?js,console)

93

```
var line = function (lineLength){        //确保 lineLength 大于或等于 0
   var line = "========================================";
   lineLength = Math.max(0, lineLength);
   lineLength = Math.min(40, lineLength); //确保 lineLength 小于或等于 40
   return line.substr(0, lineLength);    //使用 substr 字符串方法返回一条正
                                         //确长度的分隔线
}
console.log(line(30));
console.log(line(40));
console.log(line(50));
```

以上代码中使用 substr 方法来返回字符串的一部分，详细内容将在下一节中讨论。

针对各种各样常用的数学任务，JavaScript 拥有大量的 Math 方法，读者可以在以下网址查阅：www.room51.co.uk/js / math.html。

7.3.3　String 方法

针对用户想要创建的每一种字符串，JavaScript 都提供了具体的方法。 这些函数有助于程序员以各种方式操作字符串。以下代码清单 7-7 使用 toUpperCase 方法将字符串转换为大写字母，如下所示：

```
>Jupiter  becomes  JUPITER
```



代码清单 7-7　将字符串转换为大写字母 (http://jsbin.com/jizaqu/edit?js,console)

```
var planet = "Jupiter";
```

```
var bigPlanet = planet.toUpperCase();    //使用圆点运算符在 planet 字符串上
                                         //调用方法 toUpperCase
console.log(planet + " becomes " + bigPlanet);
```

如上述代码所示，我们可以使用圆点运算符在 planet 字符串上调用方法。作为一种方法，toUpperCase 能够使用其所在变量 planet 的值，因此无需将 planet 作为圆括号中的参数传递给该函数。

虽然 String 方法可以作用于其所在的变量，但这些方法依然是函数，因此也可以向它们传递参数。图 7-7 所示为 substr 方法如何使用 message 的值和两个参数的值。

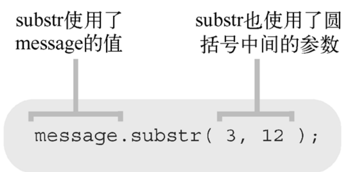


图 7-7　方法可以使用其所在变量的值以及参数的值

以下代码清单是一个使用 substr 方法的示例，在控制台上显示一个子字符串，如下所示：

94

```
>choose to go
```



代码清单 7-8　找到子字符串
(http://jsbin.com/mesisi/edit?js,console)

```
var message = "We choose to go to the Moon!";
console.log(message.substr(3, 12));
```

substr 方法接受两个参数：原始字符串中的起始位置和需要返回的字符数。当指定一个字符在字符串中的位置时，计数为零，即第一个字符是位置 0，第二个字符是位置 1，第三个字符是位置 2，以此类推（字符串从零开始计数似乎有点奇怪，但是在编程语言中这确实是很常见的做法）。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
W	e		c	h	o	o	s	e		t	o		g	o

substr（3，12）从位置 3 处的字符 c 开始，返回一个长度为 12 的字符串，即从位置 3 到位置 14。

其实，不必像这样花费大量的时间去数字符串中某字符的位置。我们可以改用 indexOf 方法来返回字符串中某个指定字符第一次出现的位置。

下一个代码清单 7-9 中使用 indexOf 来查找字符 M 在字符串中的位置，然后将位置传递给 substr 以获取长度为 3 的子字符串，在控制台上生成牛的叫声 Moo。



代码清单 7-9　使用 indexOf 查找字符
(http://jsbin.com/bidabi/edit?js,console)

```
var message = "The cow jumped over the Moon!";
```

```
var charIndex = message.indexOf("M");
console.log(message.substr(charIndex, 3));
```

此外，还可以使用 indexOf 来查找超过一个字符长度的搜索字符串。以下示例使用了代码清单 7-9 里面的变量 message 中存放的字符串：

```
message.indexOf("cow"); // Returns 4
message.indexOf("the"); // Returns 20
message.indexOf("not"); // Returns -1
```

请注意，indexOf 是区分大小写字母的，因此 the 和 The 是不同的，如果没有找到字符串，则返回-1。

如同 Math 一样，JavaScript 中有很多 String 方法，读者可以到本书的在线网站上查询，网址如下：www.room51.co.uk/js/string-methods.html。 95

那么，String 是对象吗？

在代码清单 7-7 中，将字符串“Jupiter”赋值给变量 planet， 然后调用 toUpperCase 方法：planet.toUpperCase()。

既然方法是设置为对象属性的函数，那么我们如何调用字符串上的方法？

其实，在幕后，每当我们访问一个字符串值时，JavaScript 就会创建一个特殊的对象 String 来包装这个值，该对象包括所有 String 方法。一旦执行完这个语句，该对象即被销毁。

所以，String 并不是对象，而是 JavaScript 为我们提供的一些有用的属性和方法。

7.3.4　spacer——将更多的方法收入命名空间

在第 7.3.1 节中，spacer 收入了两个函数 blank 和 newLine，下面可以再添加一些有趣的函数来使输出格式更加美观。

（1）line 是一个升级版的行分隔函数，可以返回一条指定长度的线，指定长度在 0 到 40 个字符之间，可选字符有五种：“*”、“+”、“=”、“ - ”或“ ”，如下所示：

```
spacer.line(10, "*");
spacer.line(6, "+");
    > **********
    > ++++++
```

（2）wrap 函数的返回值为指定长度的文本，该文本的首尾可以采用指定字符，如下所示：

```
spacer.wrap("Saturn", 10, "=");
spacer.wrap("Venus", 12, "-");
spacer.wrap("Mercury", 14, "+");
    > = Saturn =
    > - Venus    -
    > + Mercury    +
```

（3）box 函数的返回值为指定长度的文本，该文本的首尾为指定字符，并在上面和下面各有一行该指定字符，形状好似将该文本装在字符框内，如下所示：

```
spacer.box("Saturn", 10, "=");
spacer.box("Venus", 12, "-");
    ==========
    = Saturn =
    ==========
    ------------
    - Venus    -
    ------------
```

96

以下代码清单 7-10 是 spacer 命名空间中这些新方法的代码。请注意 spacer.wrap 使用 spacer.line 来填充文本与空格字符；spacer.box 使用 spacer.line 和 spacer.wrap 来生成其框轮廓。该代码清单可以帮助读者详细了解如何使用这些新方法。



代码清单 7-10　将若干函数作为对象的属性 (http://jsbin.com/kayono/edit?js,console)

```
var spacer ={
   blank: function (){        //返回一个空字符串
      return "";
   },
   newLine: function (){
      return "\n";            //返回一个换行符
   },
   line: function (length, character){        //返回一个指定长度的字符
      var longString ="****************************************";
      longString += "----------------------------------------";
      longString += "========================================";
      longString += "++++++++++++++++++++++++++++++++++++++++";
      longString += " ";
      length = Math.max(0, length);
      length = Math.min(40, length);
      return  longString.substr(longString.indexOf(character), length);
   },
   wrap : function (text, length, character){      //使用空格和新增加的首
                                                    //尾字符来填充文本
      var padLength = length - text.length - 3;
      var wrapText = character + " " + text;
      wrapText += spacer.line(padLength, " ");
      wrapText += character;
      return wrapText;
   },
vbox: function (text, length, character){   //将文本置于指定字符形成的框内
      var boxText = spacer.newLine();
      boxText += spacer.line(length, character) + spacer.newLine();
      boxText += spacer.wrap(text, length, character) + spacer.newLine();
      boxText += spacer.line(length, character) + spacer.newLine();
      return boxText;
```

```
    }
};
console.log(spacer.box("Mercury", 11, "=")); //测试 box 方法
console.log(spacer.box("Mars", 11, "*"));
```

小贴士

请注意 wrap 和 box 方法按照相同的顺序来使用以下几个参数：text、length 和 character。line 方法虽然只用到这三个参数中的两个，但次序未变：length 和 character。这样有意识地为参数排序可以有效降低在调用方法时的出错率。

97

7.3.5　进一步探索命名空间

命名空间中的这些新方法究竟如何运作？我们充满好奇，勇敢地深入探索一下。别忘了，我们拥有 JS Bin，在 JavaScript 面板的控制面板提示符下我们可以大胆地实验。例如，针对以下 line 方法，可以首先在控制台提示符下对变量 longString 进行声明和赋值：

```
>var longString = "**********----------==========++++++++++"
```

然后，可以试试 indexOf 方法：

```
>longString.indexOf("*")
0
>longString.indexOf("=")
20
```

放松点，别紧张。即使刚开始不理解也不必担心，因为在这些方法中出现了很多新概念，只要坚持下去，多想想就会逐渐理解。

1.　line 方法

在 wrap 和 box 方法中都使用了 line 方法。该方法返回一个指定长度的字符串，可选字符有 5 种：“*”、“+”、“=”、“ - ”或“ ”：

```
line: function (length, character) {
    var longString = "****************************************";
    longString += "----------------------------------------";
    longString += "========================================";
    longString += "++++++++++++++++++++++++++++++++++++++++";
    longString += "                                        ";
    length = Math.max(0, length);
    length = Math.min(40, length);
    return longString.substr(longString.indexOf(character), length);
},
```

该方法首先创建了一个由所有字符组成的长字符串。具体做法是一次次地使用+=运算符将新字符串附加到现有字符串，结果是变量 longString 最终成为以下模样的字符串：

```
longString = "**********----------==========++++++++++          ";
```

篇幅所限，以上缩短版的 longString 中，每种字符只显示 10 个，而实际上该方法中每种

字符会显示 40 个。该代码中还使用了字符串方法 indexOf 来查找在调用 line 时所指定的字符第一次出现的位置。下例正是使用了上面缩短版的 longString 字符串：

98

```
longString.indexOf("*"); // 0
longString.indexOf("-"); // 10
longString.indexOf("="); // 20
longString.indexOf("+"); // 30
longString.indexOf(" "); // 40
```

在已经找到指定字符首次出现的位置时，就可以使用 substr 来抓取指定长度的子字符串：

```
longString.substr(10, 6); // ------
longString.substr(30, 3); // +++
```

但是 line 方法中不能使用字面值, 需要使用形参 length 和 character，以便在调用该函数时向其传递实参：

```
// 得到第一个匹配字符的位置
var firstChar = longString.indexOf(character);
//从第一个匹配字符开始，获取所需长度的字符串
//字符
var requestedLine = longString.substr(firstChar, length);
```

以上代码中使用了更多的变量，看似可使代码更易于理解，但其实并非真正需要这些额外的变量。在以下代码中，line 找到子字符串并同时返回值：

```
return longString.substr(longString.indexOf(character), length);
```

请注意，空格是一个可用字符，可以用来填充控制台上的标题和带框形状。以下 wrap 方法就是使用 line 实现了上述效果。

2. wrap 方法

为了形成一些带框文本的中间行，wrap 方法返回一个指定长度的字符串。参数 character 指定该字符串的第一个和最后一个字符：

```
wrap : function (text, length, character) {
    var padLength = length - text.length - 3;
    var wrapText = character + " " + text;
    wrapText += spacer.line(padLength, " ");
    wrapText += character;
    return wrapText;
},
```

以下是一组由 wrap 返回的长度递增的字符串：

```
spacer.wrap("Neptune", 11, "="); // = Neptune =
spacer.wrap("Neptune", 12, "="); // = Neptune  =
spacer.wrap("Neptune", 13, "="); // = Neptune   =
```

该方法在最后一个字符的左边填充若干个空格，来确保整个字符串的长度正确（图 7-8）。 99

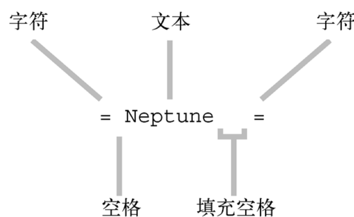


图 7-8　包裹文本由字符+空格+文本+填充空格+字符组成

要想找到填充空格的长度，请从整个字符串的所需长度入手，减去要包裹文本的长度，然后再减去三个字符，即第一个字符、空格和最后一个字符：

```
var padLength = length - text.length - 3;
```

length 属性可用于所有字符串：

```
var text = "Neptune";
text.length; // 7
```

计算好所需填充空格的长度之后，wrap 借助于 line 方法来获取指定长度的空格字符串：

```
spacer.line(padLength, " ");
```

wrap 方法通过连接所有片段（字符、空格、文本、填充空格和字符）来构建返回字符串。

3. box 方法

最后介绍 box 方法。该方法使用 line 和 wrap 将字符串包围在一个指定长度的方框中：

```
spacer.line（11， “*”）；                 //  ***********
spacer.wrap（“Neptune”， 11，“*”）；//  * Neptune *
spacer.line（11，“*”）；                  //  ***********
```

该方法中使用了 newLine，以便单个返回的字符串在控制台上可以跨行显示：

```
box: function (text, length, character) {
    var boxText = spacer.newLine();
    boxText += spacer.line(length, character) + spacer.newLine();
    boxText += spacer.wrap(text, length, character) + spacer.newLine();
    boxText += spacer.line(length, character) + spacer.newLine();
    return boxText;
}
```

非常好！现在，Spacer 命名空间已经拥有许多有用的方法，在控制台上显示格式化的信息。这些方法在 The Crypt 中可以大显身手。赶紧试试吧！ 100

7.4　The Crypt——将玩家对象作为参数

在本章中，我们知道 JavaScript 对象可以用作参数、返回值和命名空间，本节将以上知识应用于游戏 The Crypt。图 7-9 显示了本节内容的重点：通过使用带有对象的函数来显示玩家信息。

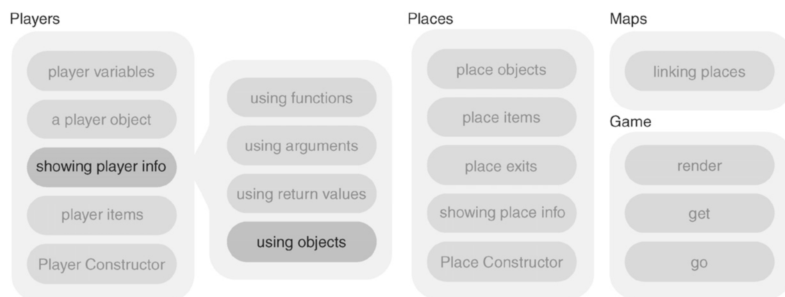


图 7-9　The Crypt 中的游戏元素

在第 6 章中，我们构建了一些函数来显示游戏中玩家的信息。在这些函数中，针对不同的玩家属性使用了不同的参数。以下是 getPlayerInfo 函数调用：

```
getPlayerInfo(player1.name, player1.place, player1.health);
```

在本章中，我们已经知道，可以将一个玩家对象直接作为参数进行传递，并让函数来选择所需的属性。getPlayerInfo 函数调用就会变成：

```
getPlayerInfo(player1);
```

显然，程序代码变得更加简洁！

我们知道，在 spacer 命名空间中有一些好用的方法可以格式化文本，例如 box 和 wrap 方法。如下所示，游戏中玩家的各种信息长度不一，地点信息字符串有可能是最长的，健康值字符串有时还会更长，如何才能找到合适的边框长度来包裹这些信息？

```
> ================================    > ++++++++++++++++++++
> = kandra is in The Dungeon of Doom =    > + Dax is in Limbo    +
> = kandra has health 50            =    > + Dax has health 40  +
> ================================    > ++++++++++++++++++++
```
101

首先，必须查明哪一行字符串是最长的。Math. max 方法可以解决这个问题：

```
var longest = Math.max（place.length，health.length）;
```

不要忘记两端的边框和空格：

```
var longest = Math.max（place.length，health.length）+ 4;
```

以下代码清单 7-11 使用 spacer 命名空间的多种方法，可以在 JS Bin 上运行。



代码清单 7-11　使用对象显示玩家信息
(http://jsbin.com/beqabe/edit?js,console)

```
var getPlayerName = function (player) {
    return player.name;
};
var getPlayerHealth = function (player) {
    return player.name + " hashealth " + player.health;
};
var getPlayerPlace = function (player) {
    return player.name + " isin " + player.place;
};
var getPlayerInfo = function (player, character) { /*找到地点字符串和健康
                                                    *值字符串两者当中哪一个
                                                    *比较长
                                                    */
    var place = getPlayerPlace(player);
    var health = getPlayerHealth(player);
    var longest = Math.max(place.length, health.length) + 4;
    var info = spacer.box(getPlayerName(player), longest, character);
    info += spacer.wrap(place, longest, character);
    info += spacer.newLine() + spacer.wrap(health, longest, character);
    info += spacer.newLine() + spacer.line(longest, character);
    info += spacer.newLine();
    return info;
};
var player1 = { name: "Kandra", place: "The Dungeon of Doom", health: 50 };
var player2 = { name: "Dax", place: "Limbo", health: 40 };
console.log(getPlayerInfo(player1, "="));//指定在显示玩家信息时要使用的字符
console.log(getPlayerInfo(player2, "+"));
```

7.5　本章小结

- 使用对象作为参数，并在函数体内访问对象的属性：

```
var getPlayerHealth = function (player) {
    return player.name + " hashealth " + player.health;
};
getPlayerHealth(player1);
```

102

- 在函数体内更新对象并添加新属性：

```
var calculateSizes = function (rectangle) {
    rectangle.area = rectangle.width * rectangle.height;
    rectangle.perimeter = 2 * (rectangle.width + rectangle.height);
};
```

- 使用关键字 return，从函数中返回新对象或现有对象：

```
var getRectangle = function (width, height) {
    return {
        width: width,
        height: height,
        area: width * height
    };
};
```

- 通过将函数设置为对象的属性来创建方法。
- 使用对象作为命名空间来收集相关的函数和属性：

```
var spacer = {};
spacer.newLine = function () {
    return "\n";
};
```

- 使用 Math 对象及其方法，如 Math.max 和 Math.min。

103 - 使用字符串的 length 属性以及 indexOf 和 substr 字符串方法。

第 8 章　数组：将数据存入列表

本章内容包括：

- 对值进行分组
- 创建数组
- 访问数组中的元素
- 操作数组中的元素
- 使用 forEach 访问每个元素

本书前几章的内容都是关于如何组织数据以及如何组织程序代码。在本章中,我们将继续这个主题，不仅将各个项目分组，而且把它们按顺序进行分组。

在游戏 The Crypt 中，玩家终于能够在探险途中收集他们所找到的物品了，正是借助数组，他们才开启了宝贝收藏之旅。

8.1　创建数组并访问元素

使用列表是编程的一个重要部分。在现实生活中，以列表形式呈现的事物数不胜数，例如博客里的文章、测验中的问题、股票价格、电子邮件、文件、推特和银行交易等等。事实上，以上刚刚读到的内容就是一个列表。在列表中，有时候顺序并不重要，但有时候顺序又很重要。例如，一个姓名列表如果只是一个团队的成员名单，并没有先后顺序；但是，如果这个姓名列表是一个比赛的排名，那么顺序是非常重要的！——博尔特（100 米赛跑冠军）必须排在第一个。与许多其他编程语言一样，在 JavaScript 中，我们也把有序列表称为数组。 104

数组中的项目称为数组元素，我们通常会以某种方式处理数组元素，例如：

（1）对每个元素执行一些操作，例如在控制台上显示该元素，或者每个元素值增加 20％。

（2）只找到符合条件的某些元素，例如 LadyGaga 的所有推文，某个月份的博客帖子，或者回答正确的问题。

（3）将所有元素合并成一个值，例如一个价格列表的总和，或每场比赛的平均得分数。

JavaScript 中的数组对象可以处理以上所有内容，而且不仅限于此。让我们从头开始，首先了解如何创建数组。

8.1.1　创建数组

要创建数组，请使用方括号。创建之后，可以将该数组赋值给一个变量，以便在代码中引用该变量，如图 8-1 所示：

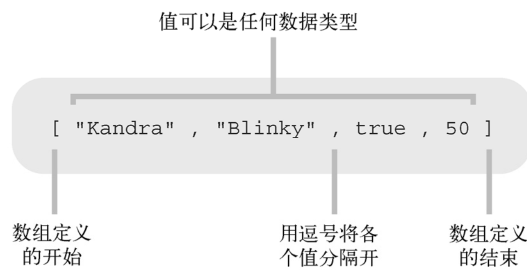


图 8-1　使用方括号创建数组

以下代码列表创建了两个数组，并在控制台上显示，输出如下：

```
> [3, 1, 8, 2]
> ["Kandra", "Dax", "Blinky"]
```



代码清单 8-1　创建数组
(http://jsbin.com/cevodu/edit?js,console)

```
var scores;
var names;
scores = [ 3, 1, 8, 2 ];      //创建一个数值数组，并将其赋值给变量 scores
names = [ "Kandra", "Dax", "Blinky" ];//创建一个字符串数组，并将其赋值给变量 names
console.log(scores);
console.log(names);
```

105

数组元素可以是数字、字符串、对象、函数、任何数据类型、混合类型甚至是数组。使用逗号来分隔各个数组元素。正如花括号代表对象，关键字 function 代表函数，JavaScript 使用方括号代表创建了一个数组。创建之后，可以将数组赋值给变量，将其设置为属性，将其包含在另一个数组中，或将其传递给函数。

代码清单 8-2 创建了对象数组，表示今年和明年分别打算访问的地方，在 JS Bin 上的显示如图 8-2 所示。



图 8-2　JS Bin 显示代码清单 8-2 中的每一个数组



代码清单 8-2　在数组中使用已定义的对象
(http://jsbin.com/gizulu/edit?js,console)

```
var place1 = { name : "The Pyramids", country : "Egypt" };
var place2 = { name : "The Grand Canyon", country : "USA" };
var place3 = { name : "Bondi Beach", country : "Australia" };
var thisYear = [ place1, place2 ]; //创建一个对象数组，并将其赋值给变量thisYear
var nextYear = [ place3 ];       //在数组中可以有零个或多个元素
console.log(thisYear);
console.log(nextYear);
```

8.1.2　访问数组元素

既然可以创建一个数组并将其赋值给一个变量，就可以将该变量传递给一个函数，例如 console. log。正如我们需要剥开橙子皮才能萃取里面的橙汁，有时，我们也需要访问数组内部的某个元素。这时，可以使用方括号，它具有以下双重作用：在定义数组时，方括号用于包裹列表；另一方面，方括号还可以用于访问数组内的单个元素。

如图 8-3 所示，我们使用索引值来标识每一个元素，表示每一个元素在列表中的位置。 106
数组中第一个元素的索引值为 0，第二个元素的索引值为 1，依此类推。我们可以把索引值看作是距离数组开始点的偏移量：第一个元素距离开始点为 0，第二个元素距离开始点为 1，依此类推。

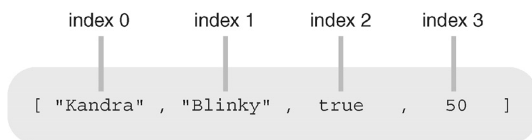


图 8-3　数组中的每一个元素都有一个索引值，从 0 开始

要通过某一个指定索引值来获取元素的位置，请将索引值置于方括号内，方括号的位置在变量名之后，如图 8-4 所示。



图 8-4　使用方括号和索引值来访问已赋值给变量的数组内部元素

在以下示例中，我们创建了一个数组，并将其赋值给一个变量：

```
var scores;
scores = [ 3, 1, 8, 2 ];
```

要获取数组中第三项的值：8，请将索引值 2（因为索引值是从 0 开始的）放在变量名后的方括号内：

```
scores[2];
```

我们可以将 scores[2]赋值给另一个变量，将其设置为对象的属性，将其用于表达式中，

或者将其作为参数传递给一个函数。代码清单 8-3 使用 scores 数组中元素的值创建字符串，并在控制台上显示如下：

```
> There are 4 scores:
> The first score is 3
> The second score is 1
> The third score is 8
> The fourth score is 2
```

107 显然，代码中还使用了数组的 length 属性，显示该数组中元素的数量为 4。



代码清单 8-3　访问数组元素
(http://jsbin.com/qemufe/edit?js,console)

```
var scores = [ 3, 1, 8, 2 ];
console.log("There are " + scores.length+ " scores:");  //使用 length 属
                                                    //性来显示数组中元素的数量
console.log("The first score is " + scores[0]);     //数组中第一个元素的索
                                                    //引值为 0
console.log("The second score is " + scores[1]);
console.log("The third score is " + scores[2]);
console.log("The fourth score is " + scores[3]);//数组中最后一个元素的索引
                                                //值为元素的数量减 1
```

我们假设有一个 day 数组，数组元素是五个工作日的名称，即星期一、星期二……星期五，如下所示。现在想要得到一个星期中某一天的名称，例如，第四天是星期几？

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
```

我们发现，JavaScript 中的索引值与元素的名称之间存在不匹配。例如，一周的第 1 天索引值为 0，一周的第 4 天索引值为 3。为了消除这种别扭的感觉，可以使用变量 dayInWeek 作为数组的索引值，如下所示：

```
// 我想要得到一周中的第四个工作日
dayInWeek = 4;
```

但是，问题并没有完全解决。使用 dayInWeek 作为数组的索引值，却得到了一个错误的元素，由索引值 4 得到数组中第 5 个元素：Friday。

以下代码清单 8-4 中暴露并解决了这个问题，在控制台上显示了两行信息，第一行为错误信息，第二行为正确信息，如下所示：

```
> Friday
> Thursday
```



代码清单 8-4　使用变量作为索引
(http://jsbin.com/veyexa/edit?js,console)

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
```

```
var dayInWeek = 4; //使用dayInWeek变量作为索引值
console.log( days[dayInWeek] );
console.log( days[dayInWeek - 1] );//从dayInWeek中减1，以访问到正确的元素
```

在以上代码中，第一次调用 console.log 会显示错误的一天，因为变量 dayInWeek 没有考虑到数组是从 0 开始的；索引值从 0 开始，而不是从 1 开始。第二次调用 console.log，从 dayInWeek 中减 1 就解决了上述问题，即一周中第 4 天的索引值为 3。 108

问题得以解决！让我们保持良好势头继续前进，攻克下一个堡垒。当地铅笔博物馆每天都在记录访客的数量，他们想编写一个程序，实现以下功能：针对存放在数组中的一周游客数量，程序能够显示指定日的游客数量：

```
> There were 132 visitors on Tuesday
```

我们决定创建一个函数 getVisitorReport 来生成报告并返回值。然后，可以选择把该报告显示在控制台、网页或电子邮件中。在以下代码清单 8-5 中，将生成星期二的报告并显示在控制台上。



代码清单 8-5　将数组传递给函数
(http://jsbin.com/bewebi/edit?js,console

```
var getVisitorReport = function (visitorArray, dayInWeek){   //包括两个
                                                         //形参：visitorArray用于传
                                                         //递访客数量，dayInWeek用于
                                                         //传递报告中是星期几
    var days = [
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday"
    ];
    var index = dayInWeek - 1;
    var visitorReport;
    visitorReport = "There were ";
    visitorReport += visitorArray[index]; //使用+=运算符来更新
                                          //visitorReport：将更多的
                                         //文本连接到visitorReport当前值
    visitorReport += " visitors ";
    visitorReport += "on " + days[index];
    return visitorReport;      //从函数返回结果
};
var visitors = [ 354, 132, 210, 221, 481 ];
var report = getVisitorReport(visitors, 2);   //获取星期二的报告并将其赋值
                                              //给变量report
console.log(report);
```

在以上代码中，创建了一个访客数量的数组，赋值给变量 visitor，并将其作为参数传递

给函数 getVisitorReport。在函数中将数组赋值给变量 visitorArray，用于生成报告。函数 getVisitorReport 返回报告，并赋值给变量 report，显示在控制台上。函数中的+=运算符将其
109 右侧的值添加给其左侧的变量。此例中，已将字符串赋值给 visitorReport 变量，所以+=将其右侧的值连接到其左侧的变量上。

如上所示，我们能够创建一个数组并访问它的元素。当数组中存有数据时，我们就需要使用一些方法对数组进行相应的操作，下面就来学习几种最常用的数组处理方法。

8.2 数组方法

数组是 JavaScript 语言提供的一种对象类型，用于管理代码清单。JavaScript 还提供了许多可以与数组一起使用的函数。我们还记得，当函数赋值给一个对象的属性时，我们把这些函数称作对象的方法，既然数组也是一种对象类型，那么它们的函数也可以称作方法。

在本节中，将看到只有几个方法可以与数组一起使用：push、pop 和 splice 用于添加和删除元素，slice 用于抓取连续的元素，join 用于连接数组元素来形成一个字符串，forEach 用于将每个元素作为参数传递给指定函数，见表 8-1。欲了解更多数组方法和示例，请访问本书的相关网址：www.room51.co.uk/js/array-methods.html。

表 8-1 数组方法

方　法	含　义	示　例
push	将元素追加到数组的末尾	items.push("Putmelast");
pop	从数组末尾删除一项	wasLast = items.pop();
join	连接数组中的所有元素，在每两个元素之间插入一个字符串	allItems = items.join(",");
slice	从现有数组中的一系列元素截取创建一个新数组。传入截取起始位置的索引值和终止位置的索引值	section = items.slice(2, 5);
splice	通过添加和/或删除连续元素来更改数组。传入开始删除元素的索引值，要删除元素的数量和想要添加的元素	out = items.splice(1,2,"new");
forEach	将每个元素依次传递给指定函数	items.forEach(function (item){ console.log(item); }

110 下面，我们来学习 push、pop 和 join 三个数组方法。

8.2.1 添加和删除元素

在以下代码清单 8-6 中，首先，创建一个空数组，并将其赋值给变量 items；然后，使用 push 方法将三个元素添加到该数组，将完整数组显示在控制台上，如下所示：

```
> ["The Pyramids", "The Grand Canyon", "Bondi Beach"]
```

之后，再使用 pop 方法删除最后一项并显示；最终，将整个数组再次显示在控制台上，这时各个元素连接在一起形成一个字符串，如下所示：

```
>Bondi Beach was removed
> The Pyramids and The Grand Canyon
```



代码清单 8-6　使用 push、pop 和 join 对数组进行操作 (http://jsbin.com/faqabu/edit?js,console)

```
var items = [];          //创建一个空数组，并将其赋值给变量 items
var item = "The Pyramids";
var removed;
items.push(item);        //将 item 的值加到该数组的末尾
items.push("The Grand Canyon");    //将字符串加到该数组的末尾
items.push("Bondi Beach");
console.log(items);
removed = items.pop();             //删除最后一项，并将其赋值给变量 removed
console.log(removed + " was removed");
console.log(items.join(" and "));  //使用 join 方法连接数组中的所有元素，在两
                                   //个元素之间插入字符串" and "
```

在 JavaScript 中，可以使用 push、pop 和 join 函数作为每个数组的属性。既然它们是数组的属性，就可以使用圆点运算符对其进行访问，因此可以使用圆点运算符来调用函数，如 items.push（itemToAdd）。

8.2.2　截取和拼接数组

为了演示 slice 和 splice 这两种数组方法，我们继续使用上面的度假地数组，并将结果展示在 JS Bin 控制台上。我们知道，在控制台上，输入回车键〈Enter〉表示要执行一条新的语句，但是，如果同时按下 Shift 和 Enter 键，就会实现一条命令中的换行，而不是执行一条新的语句，因此，一条命令可以跨行显示。如果不小心按〈Enter〉键执行了一条未完成的语句，可以按键盘上的向上箭头来检索前面各条语句。在控制台上，用户输入的每一条语句都是以>作为起始来显示的。控制台自动显示每个调用函数的返回值，并没有>作为起始，下文中用粗体来显示这些返回值： 111

```
>var items = [
"The Pyramids",
"The Grand Canyon",
"Bondi Beach",
"Lake Garda"]
undefined
>items
["The Pyramids", "The Grand Canyon", "Bondi Beach", "Lake Garda"]
```

slice 方法只是截取原始数组的一部分构成一个新数组，对原始数组不做更改。slice 方法有两个参数，第一个参数是截取所需第一个元素的索引值，第二个参数是截取不需要的第一个元素的索引值，如下所示。别忘了，第一个元素的索引值为 0。

```
>items.slice(0, 2)
["The Pyramids", "The Grand Canyon"]
>items.slice(2, 3)
["Bondi Beach"]
```

items.slice(2, 3)表明截取需要索引值 2 以后的元素，但不需要从索引值 3 开始的后续元素，换句话说，只需要索引值为 2 的元素。

如果需要第一个参数之后的所有元素，可以省略第二个参数。如果想要整个数组，就省略两个参数，如下所示：

```
>items.slice(2)
["Bondi Beach", "Lake Garda"]
>items.slice()
["The Pyramids", "The Grand Canyon", "Bondi Beach", "Lake Garda"]
```

splice 方法确实对原始数组有所改变，不但允许从数组中删除项目，而且可以加入新项目。要删除项目，需要两个参数，第一个参数是你想要删除的第一个元素的索引值，第二个参数是你想要删除元素的数量。该方法的返回值是由这些被删除元素所组成的新数组，如下所示：

```
>items.splice(2, 1)
["Bondi Beach"]
>items
["The Pyramids", "The Grand Canyon", "Lake Garda"]
>items.splice(0, 2)
["The Pyramids", "The Grand Canyon"]
>items
["Lake Garda"]
```

112

想要将新元素插入某一数组，请将新元素作为第三个参数添加到以上两个参数之后。在以下示例中，不删除任何项目：

```
>items.splice(0, 0, "The Great Wall", "St Basil's")
[]
>items
["The Great Wall", "St Basil's", "Lake Garda"]
```

在以下示例中，删除了一个项目：

```
>items.splice(1, 1, "Bondi Beach", "The Grand Canyon")
["St Basil's"]
>items
["The Great Wall", "Bondi Beach", "The Grand Canyon", "Lake Garda"]
```

在游戏 The Crypt 中，我们将使用 slice 和 splice 来操作各个玩家和地点项目。

8.2.3 使用 forEach 访问每一个元素

如果只是想在控制台上显示一系列项目，我们可以手动一次次地调用函数：

```
showInfo(items[0]);
showInfo(items[1]);
showInfo(items[2]);
```

但是，我们通常并不能提前知道列表中会有多少个项目，所以不能对 showInfo 的调用次数进行提前硬编码。另外，随着元素数量的增加，我们就不想再用手动方式去一次次地调用函数了。

因此，需要一种方法，使 JavaScript 能够为列表中的每一个元素调用函数，不论元素的数目有多大都没有关系。这正是 forEach 方法的作用所在。以下示例中，使用 forEach 方法为 items 数组中的每个元素调用 showInfo，这样也就不需要手动一次次地调用函数了：

```
items.forEach(showInfo);
```

forEach 方法用于实现数组的遍历，即依次把每个元素作为参数传递给括号中的指定函数，如图 8-5 所示。

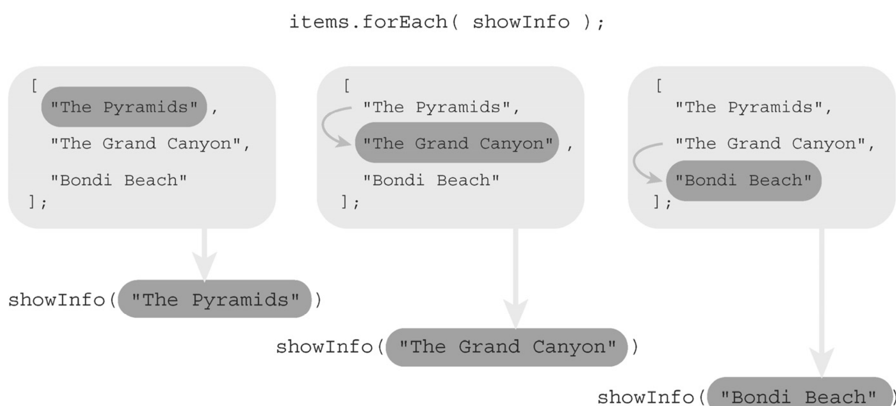


图 8-5　items.forEach 将 items 数组中的每个元素传递给函数 showInfo

以下代码清单 8-7 显示了 forEach 是如何运行的，即如何将 items 数组的各元素显示在控制台上，如下所示：

```
> The Pyramids
> The Grand Canyon
> Bondi Beach
```

113



代码清单 8-7　使用 forEach 遍历一个数组 (http://jsbin.com/sokosi/edit?js,console)

```
var items;           //使用关键字 var 声明两个变量
var showInfo;
items = [            //使用方括号创建数组，并将其赋值给变量 items
    "The Pyramids",
    "The Grand Canyon",
    "Bondi Beach"
];
showInfo = function (itemToShow) {      //使用关键字 function 创建函数，并
                                        //将其赋值给变量 showInfo
    console.log(itemToShow);
```

```
};
items.forEach(showInfo);            //为该数组中的每个元素调用函数 showInfo
```

在代码清单 8-7 中，将显示每个项目的函数赋值给变量 showInfo，然后将变量 showInfo 作为参数传递给 forEach。

如果只是一次性地使用一个函数并将其作为 forEach 的参数，就可以创建该函数并将其直接传递给 forEach，而无需一个像 showInfo 这样的额外变量。在以下代码清单 8-8 中，将函数定义直接传递给 forEach，还添加了额外的信息使输出样式更加美观，如下所示：

114

```
> Dream destinations:
> - The Pyramids
> - The Grand Canyon
> - Bondi Beach
```



代码清单 8-8　使用内嵌函数调用 forEach (http://jsbin.com/yapecu/edit?js,console)

```
var items = [ "The Pyramids", "The Grand Canyon", "Bondi Beach" ];
console.log("Dream destinations:");
items.forEach(function (item) {          //将内嵌函数传递给 forEach 方法
    console.log(" - " + item);
});
```

forEach 方法实际上将三个参数传递给指定的函数：元素、当前元素的索引值和整个数组。如果在函数定义中包含更多的形参，就可以得到更多的实参来传递给 forEach，如下所示：

```
items.forEach(function (item, index, wholeArray) {
    // item 是当前传递给函数的元素
    // index 是当前元素的索引值
    // wholeArray 与 items 完全相同
});
```

以下代码清单 8-9 显示了这三个参数，使用 forEach 将 players 数组中的每个玩家传递给函数 showArguments，得到以下输出：

```
> Item: Dax
> Index: 0
> Array: Dax,Jahver,Kandra
> Item: Jahver
> Index: 1
> Array: Dax,Jahver,Kandra
> Item: Kandra
> Index: 2
> Array: Dax,Jahver,Kandra
```



代码清单 8-9　使用由 forEach 传来的参数
(http://jsbin.com/suvegi/edit?js,console)

```
var players;
var showArguments;
players = ["Dax", "Jahver", "Kandra"];
showArguments = function (item, index, wholeArray){    //包含更多的形参来得到
                                                       //由forEach传来的实参
    console.log("Item: " + item);
    console.log("Index: " + index);
    console.log("Array: " + wholeArray);
};
players.forEach(showArguments);
```

forEach 方法其实是在调用函数。在代码清单 8-9 中，forEach 方法为 players 数组中的每个元素调用函数 showArguments。尽管并不是必须使用三个参数，但它总是将三个参数全部传递给它所调用的函数。 115

我们也可以不使用变量，直接在数组上调用像 forEach 这样的数组方法。以下代码清单 8-10 是对代码清单 8-9 的改写，在代码清单 8-10 中就没有将数组或函数赋值给变量。



代码清单 8-10　使用由 forEach 传来的参数——紧凑版
(http://jsbin.com/pagahe/edit?js,console)

```
["Dax", "Jahver", "Kandra"].forEach(function (item, index, wholeArray) {
    console.log("Item: " + item);
    console.log("Index: " + index);
    console.log("Array: " + wholeArray);
});
```

如果我们只是一次性地使用一个数组和函数，如代码清单 8-10 所示的紧凑型语法似乎更加可取。但是，代码清单 8-9 中的长版本代码更加易于理解，所以如果该段代码的意思在整个程序中不是非常清晰明了，最好还是选择长版本。显然，采用

```
players.forEach(showScore);
```

这样的语句能够帮助自己和其他程序员更好地理解代码。

为了进一步了解如何使用索引值参数，请去逛逛代码清单 8-11 里的商店（既然打算继续冒险，总需要为探险多买些装备）。这次需要购买四种物品，单价各不相同。该程序用于计算和显示总价，如下所示：

```
> The total cost is $41.17
```

在代码清单 8-11 中，使用两个数组，一个数组用于存放购买每种物品的数量，另一个数组用于存放每种物品的单价。



代码清单 8-11　计算购物的总价
(http://jsbin.com/zizixu/edit?js,console)

```
var getTotalBill = function (itemCosts, itemCounts){
```

```
    var total = 0;
    itemCosts.forEach(function (cost, i){      //在函数定义中包含一个额外的
                                               //形参 i，用来传入索引值
        total+= cost * itemCounts[i];          //使用+=运算符将计算结果加到
                                               //变量 total 的值上
    });
    return total;
};
var costs = [ 1.99, 4.95, 2.50, 9.87 ];
var numOfEach = [ 2, 1, 5, 2 ];
```

116

```
console.log("The total cost is $" + getTotalBill(costs, numOfEach));
```

代码清单 8-11 中，使用索引值来匹配当前所购物品的单价和物品的数量。为了匹配无误，两个数组必须采用相同的次序。读者应该已经注意到了，i 并不是一个合适的变量名称！但大多数程序员都喜欢使用 i、j 和 k 来代表索引值变量，使输入更加简捷，因此已经成为一个成熟的惯例，当他人阅读你的代码时，也会自然而然地把它们看作是计数器或索引。如果你不愿遵此惯例，就是喜欢将它们命名为 index 或 itemIndex，或其他名称，也完全没有问题。

作为本节内容的最后一个示例，让我们再次回到测验问题。针对每一道选择题，都需要给应试者呈现一个选项列表（例如 A、B、C、D），这看起来很适合采用数组和 forEach 方法，见以下代码清单 8-12。我们甚至还可以使用由多个问题选项对象构成的数组。下面先以一个问题为例，问题和选项显示如下：

```
> What is the capital of France?
> A - Bordeaux
> B - F
> C - Paris
> D - Brussels
```



代码清单 8-12　显示一道选择题
(http://jsbin.com/lobahu/edit?js,console)

```
var displayQuestion = function (questionAndAnswer){ //包含一个形参，用于传入
                                                    //questionAndAnswer 对象
    var options = [ "A", "B", "C", "D", "E" ];
    console.log(questionAndAnswer.question);  //通过参数访问
                                              //questionAndAnswer 对象的属性
    questionAndAnswer.answers.forEach(        //使用 forEach 访问分配给 answers
                                              //属性的数组内的每个元素
        function (answer, i){
            console.log(options[i] + " - " + answer);
        }
    );
};
var question1 = {
    question : "What is the capital of France?",  //将数组分配给 answers 属性
```

```
    answers : [
    "Bordeaux",
    "F",
    "Paris",
    "Brussels"
    ],
    correctAnswer : "Paris"
};
displayQuestion(question1);
```

本书第二部分将深入探究与用户的互动，届时读者会看到用户如何回答问题。 117

8.3 The Crypt——玩家的物品数组

在本节中，我们将 JavaScript 数组知识应用到游戏 The Crypt 中。图 8-6 显示了本节的重点：使用数组显示玩家的物品。

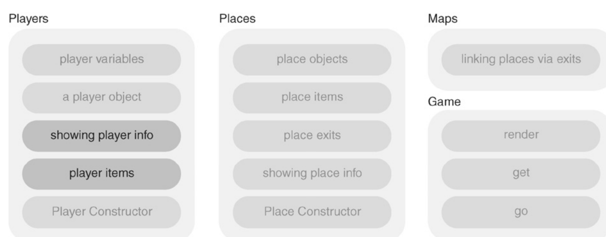


图 8-6　The Crypt 中的游戏元素

在本书的第一部分，已经介绍了 JavaScript 的一些核心概念，使用这些概念，我们可以在游戏 The Crypt 中创建和使用玩家信息。具体来说，我们已经可以使用变量来存储和检索玩家信息，已经可以使用对象将玩家属性集合在一起，当然，我们也应该可以使用数组将玩家收集的物品放入一个列表，使用数组方法来添加和删除列表中的元素，然后使用函数来显示每个玩家的信息。

在以下代码清单 8-13 中，将以上所有概念结合在一起：创建一个玩家，显示玩家信息，拾取一个新的物品，以及显示更新的信息。我们还可以使用第 7 章 spacer 命名空间对输出内容进行格式化处理，输出效果如图 8-7 所示。

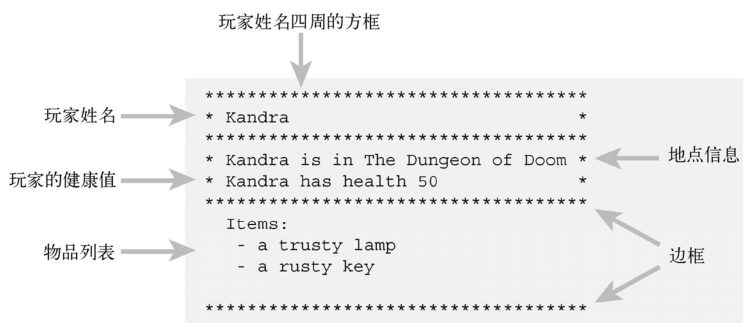


图 8-7　在控制台上显示的玩家对象的各种信息 118

为了节省版面，在代码清单 8-13 中省略了 spacer（因为它没有任何变化），但是在 JS Bin 中并没有省略。此代码清单以第 7 章中代码清单 7-11 为基础，添加了一项新功能：列出玩家的物品。



代码清单 8-13　显示玩家的物品
(http://jsbin.com/mecude/edit?js,console)

```
var getPlayerName = function (player) {
   return player.name;
};
var getPlayerHealth = function (player) {
   return player.name + " has health " + player.health;
};
var getPlayerPlace = function (player) {
   return player.name + " is in " + player.place;
};
var getPlayerItems = function (player) {      //定义函数来构建一个字符串，
                                              //用于显示玩家的物品
   varitemsString = "Items:" + spacer.newLine();
   player.items.forEach(function (item) {     //使用 forEach 将 items 数组
                                              //的每个元素附加到该字符串上
      itemsString += " - " + item + spacer.newLine();
   });
   return itemsString;
};
var getPlayerInfo= function (player, character) {
   var place = getPlayerPlace(player);
   var health = getPlayerHealth(player);
   var longest = Math.max(place.length, health.length) + 4;
   var info = spacer.box(getPlayerName(player), longest, character);
   info += spacer.wrap(place, longest, character);
   info += spacer.newLine() + spacer.wrap(health, longest, character);
   info += spacer.newLine() + spacer.line(longest, character);
   info += spacer.newLine();
   info += " " + getPlayerItems(player);//调用 getPlayerItems，将 items
                                         //字符串包含在玩家信息中
   info += spacer.newLine();
   info += spacer.line(longest, character);
   info += spacer.newLine();
   return info;
};
var showPlayerInfo = function (player, character){  //创建函数 show
                                                    //PlayerInfo，将信息字
                                                    //符串显示在控制台上
   console.log(getPlayerInfo(player, character));
};
```

```
var player1 = {
    name: "Kandra",
    place: "The Dungeon of Doom",
    health: 50,
    items : ["a trusty lamp"] //给玩家一件物品
};
showPlayerInfo(player1, "="); //传入玩家和边框字符作为实参，调用 showPlayerInfo
player1.items.push("a rusty key");        //使用 push 在 items 数组中添加一件
                                          //额外的物品
showPlayerInfo(player1, "*");
```

119

现在，Player 对象包含了一个 items 数组：

```
var player1 = {
    name: "Kandra",
    place: "The Dungeon of Doom",
    health: 50,
    items: ["a trusty lamp"]
};
```

在以上代码中，items 数组最初只有一个元素，通过使用 push 数组方法为其添加了一个新元素：

```
player1.items.push("a rusty key");
```

getPlayerItems 使用 forEach 将 items 数组中的每个元素传递给一个函数。该函数使用+=将每一件物品添加到字符串，从而构建了一个玩家全部物品的列表：

```
player.items.forEach(function (item) {
    itemString += " - " + item + spacer.newLine();
});
```

函数 showPlayerInfo 调用 getPlayerInfo 来检索玩家信息字符串，然后在控制台上显示信息。

现在我们已经学会处理显示一个玩家的信息。下面，我们打算构建多个函数的集合来显示多个玩家的信息。如何用心把多个函数组织到一起？我们可以将它们收集到一个命名空间中。既然它们都与玩家相关，是否可以考虑把它们包含在 Player 对象中？针对多个类似对象的创建问题，JavaScript 提供了一种简化方法：构造函数。这就是下一章的内容，在第 9 章中，我们将构建玩家在该游戏中所探索的地点信息。

8.4　本章小结

■ 创建一个数组，使用逗号将方括号之间的各个值进行分隔：

```
[ "Kandra", "Dax", true, 50 ]
```

■ 将数组赋值给变量，然后通过在变量名后的方括号中的索引值来访问数组元素。以

下代码会显示该数组中的“Dax”元素：

120

```
var items = [ "Kandra", "Dax", true, 50 ];
console.log(items[1]);
```

- 千万别忘记，数组元素的索引值是从 0 开始的，items[1]引用 items 数组中的第二个元素。读者可以将索引值视为与数组开头的偏移量：第一个元素就是数组的开头，因此偏移量为 0；第二个元素与数组开头的偏移量为 1，依此类推。
- JavaScript 为我们提供了常用的数组方法，用于添加、删除、连接和遍历数组元素。
121 本章介绍的数组方法有 push、pop、join、slice、splice 和 forEach。

第 9 章　构造函数：构建带有函数的对象

本章内容包括：

- 使用函数创建对象
- 关键字 new
- 使用构造函数创建对象
- 特殊变量 this
- 玩家构造函数和地点构造函数

程序中常常需要创建许多彼此相似的对象，例如一个博客里可能有数百个帖子，一个日历里可能有数千个事件。相应地，程序中也需要有函数来处理这类对象。我们不再需要像第 3 章那样，用手动方式使用花括号创建每一个对象，下面我们使用函数来简化上述过程。

我们想要把：

```
planet1 = {
    name: "Jupiter",
    position: 8,
    type: "Gas Giant"
};
showPlanet = function (planet) {
    var info = planet.name + ": planet " + planet.position;
    info += " - " + planet.type;
    console.log(info);
};
showPlanet(planet1);
```

122

改成：

```
planet1 = buildPlanet("Jupiter", 8, "Gas Giant");
planet1.show();
```

在以上代码中，buildPlanet 函数创建了一个行星对象，并自动添加了 show 方法。在 buildPlanet 函数体中为对象定义了一个蓝图，然后在需要时使用该蓝图就可以生成一个新对象。这样做，有利于将这一段对象创建代码固定保持在程序中的某一个地方，有助于理解、维护和使用该代码。

可见，我们可以用一个简单的函数创建许多相似的对象。其实，还可以采用更好的方法，就是使用构造函数。正因为我们常常需要编写函数来创建对象，JavaScript 中有一个内置的简化方法，我们可以使用它来创建和显示行星对象，如下所示：

```
planet1 = new Planet("Jupiter", 8, "Gas Giant");
planet1.show();
```

构造函数将对象创建变为标准化流程，具有诸多优点：构造函数提供了一种识别对象的方法，使我们更容易区分开行星、玩家、帖子和地点对象；构造函数还提供了一种方法，使我们能够使用原型（详见本书第四部分）在多个对象之间共享某个函数。

在第 9.1 节中，将介绍如何自己编写函数来构建对象；在第 9.2 节中，将探究如何使用构造函数来简化构建对象的生产流程。现在，我们开始让生产线转动起来！

9.1 使用函数构建对象

我们不再采用手动方式使用花括号来构造每个对象，而是创建一个函数来干这种苦力活儿。我们只要向函数传递它所需要的信息，函数就会为我们返回一个崭新的对象！在大型程序中，我们可能会在代码中的多个位置创建一些类似的对象。如果拥有这样一个可以多次调用的函数，就可以避免一次次的重复创建工作，而且当我们以后需要对象有所变化时，修改工作也会变得轻松许多。

9.1.1 节展示了一个简单的对象创建函数，9.1.2 节在此基础上，为创建的对象添加了一
123 个方法。

9.1.1 添加属性

图 9-1 显示了一个 buildPlanet 函数，它可以帮助我们创建行星对象。我们只需将某一行星数据作为参数传递给该函数，就会返回一个行星对象，该对象具有以上参数的属性。我们刚刚开始学习 JavaScript，就已经可以创建行星了！

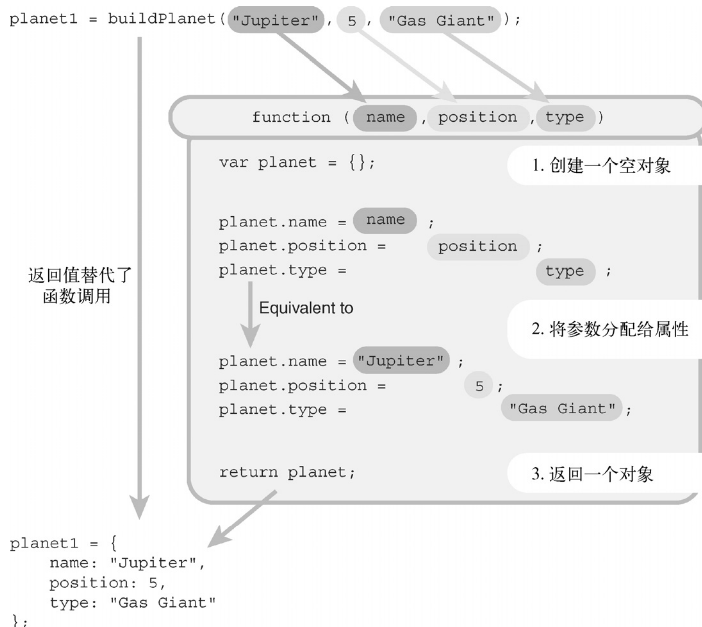


图 9-1　buildPlanet 函数创建并返回一个对象

以下代码清单 9-1 显示了 buildPlanet 函数的代码。请注意该函数如何始于创建对象，止于返回对象。



代码清单 9-1　用函数来创建对象
(http://jsbin.com/jiroyo/edit?js,console)

```
var buildPlanet = function (name, position, type){
    var planet = {};              //创建一个空对象，为正在构造的新行星做准备
    planet.name = name;           //使用传入的参数在行星对象上设置属性
    planet.position = position;
    planet.type = type;
    return planet;                //返回对象，该对象就是创建完成的新行星
};
var planet1 = buildPlanet(        //调用 buildPlanet 函数并将返回的对象赋值给
                                  //planet1 变量
    "Jupiter",
    5,
    "Gas Giant"
);
console.log(planet1.name);
console.log(planet1.position);
console.log(planet1.type);
```

124

在以上代码中，buildPlanet 函数所采用的步骤非常有助于我们理解 JavaScript 构造函数如何简化对象创建的过程，详见第 9.2 节中的构造函数。在此之前，我们首先需要学会使用一些方法来武装我们刚刚创建的对象。

9.1.2　添加方法

除了设置一些初始属性，buildPlanet 还可以为对象添加方法。请注意，方法其实是一个函数，该函数被赋值给对象的属性。我们并不是定义一个外部函数来显示关于每个行星对象的信息，而是将函数嵌入到对象内部。

在以下代码清单 9-2 中，函数返回对象之前，为每一个行星对象添加了一个 showPlanet 方法，输出如下所示：

```
> Jupiter: planet 5 - Gas Giant
```



代码清单 9-2　为函数创建的对象添加方法
(http://jsbin.com/zogure/edit?js,console)

```
var buildPlanet = function (name, position, type) {
    var planet = {};        //创建一个空对象，为正在构造的新行星做准备
    planet.name = name;     //使用传入的参数在行星对象上设置属性
    planet.position = position;
    planet.type = type;
    planet.showPlanet = function (){ //定义显示函数并将其赋值给 showPlanet 属性
        var info = planet.name;
```

```
            info += ": planet " + planet.position;
            info += " - " + planet.type;
            console.log(info);
        };
        return planet;  //返回对象，该对象就是我们创建完成的新行星
125 };
    var planet1 = buildPlanet(
        "Jupiter",
        5,
        "Gas Giant"
    );
    planet1.showPlanet();    //使用圆点运算符和圆括号调用 showPlanet 方法
```

在以上 showPlanet 方法中，构建了一个包含行星属性（name、position 和 type）的信息字符串，并将该字符串记录在控制台上。我们可以使用+=运算符将字符串附加到 info 变量上。

我们可以针对对象需要的功能为其提供尽可能多的方法。每次调用 buildPlanet 时，都会返回一个对象，该对象具有在程序中执行任务所需的属性和方法，也就是说，这个函数能够构造出各种人类想要的行星对象。当然，我们也不能太贪心，不是吗？

以下代码清单 9-3 使用 buildPlanet 函数构造了三个行星（好吧，虽然还算不上是一条生产线，但就是这个思路！），并在控制台上显示它们的详细信息：

```
> Jupiter: planet 5 - Gas Giant
> Neptune: planet 8 - Ice Giant
> Mercury: planet 1 - Terrestrial
```



代码清单 9-3　由三个构造对象组成的数组
(http://jsbin.com/jiweze/edit?js,console)

```
var buildPlanet = function (name, position, type){  //定义一个与代码清单 9-2
                                                    //相同的buildPlanet 函数
    var planet = {};
    planet.name = name;
    planet.position = position;
    planet.type = type;
    planet.showPlanet = function () {
        var info = planet.name;
        info += ": planet " + planet.position;
        info += " - " + planet.type;
        console.log(info);
    };
    return planet;
};
var planets = [    //创建一个行星对象的数组，并赋值给 planets 变量
    buildPlanet( "Jupiter", 5, "Gas Giant" ),
    buildPlanet( "Neptune", 8, "Ice Giant" ),
    buildPlanet( "Mercury", 1, "Terrestrial" )
```

```
];
planets.forEach(function (planet) {      //使用 forEach 将每个行星对象传递到
                                         //showPlanet 调用函数
    planet.showPlanet();
});
```

126

以上代码清单 9-3 创建了一个包含三个元素的数组，这三个数组元素就是三次分别调用 buildPlanet 的返回对象。将该数组赋值给 planets 变量，然后使用 forEach 遍历该数组，这样一来，就不需要将三个行星分别赋值给三个变量，而只需通过数组索引值就可以访问各个对象（谨记第一个行星的索引值为 0），如下所示：

```
planets[0].name // "Jupiter"
planets[2].type // "Terrestrial"
```

在定义 buildPlanet 时，我们将对象创建移动到一个函数中，一旦设置好对象的属性，该函数就会为我们返回一个崭新的行星。由此可见，我们自己编写的函数可以为我们创建和返回对象。比这更令人高兴的是，JavaScript 本身就可以为我们创建和返回对象，如何实现？请阅读下一节内容。

9.2　使用构造函数构建对象

在上一节中，我们自己编写函数构建对象并为对象添加方法，切实感受了对象生产线的大致模样。其实，用这种方式来构建对象的做法非常多见，以至于 JavaScript 为程序员提供了标准化生产线——构造函数。

在上一节的 buildPlanet 函数中，我们先创建了一个空对象，然后为其设置属性，并返回对象，如下所示：

```
var buildPlanet = function (name, position, type) {
    var planet = {};                  //创建一个空对象
    planet.name = name;               //
    planet.position = position;       //设置属性
    planet.type = type;               //
    return planet;                    //返回对象
};
```

其实，JavaScript 可以帮我们免费搞定这一切。使用构造函数，它将创建空对象并返回对象，我们依然可以设置属性，但其余步骤都是自动完成的。到底是什么秘密武器能够将普通函数转换为构造函数？答案是两个关键字：this 和 new。

9.2.1　构造函数

在 JavaScript 中，定义构造函数与定义其他函数没有什么差别，只是在调用构造函数时需要在前面加一个关键字 new。假设我们已经拥有一个 Planet 函数，就可以采用以下方式创建一个新的行星对象：

```
planet1 = new Planet("Jupiter", 5, "Gas Giant");
planet2 = new Planet("Neptune", 8, "Ice Giant");
```

从上例可见，欲将 Planet 函数作为构造函数使用，只需在调用 Planet 之前添加 new 关键字即可。作为惯例，构造函数名称的首字母通常采用大写字母，以便程序员容易辨识某函数为构造函数，在调用构造函数时就会使用关键字 new。图 9-2 显示构造函数如何自动创建并
127 返回一个对象。

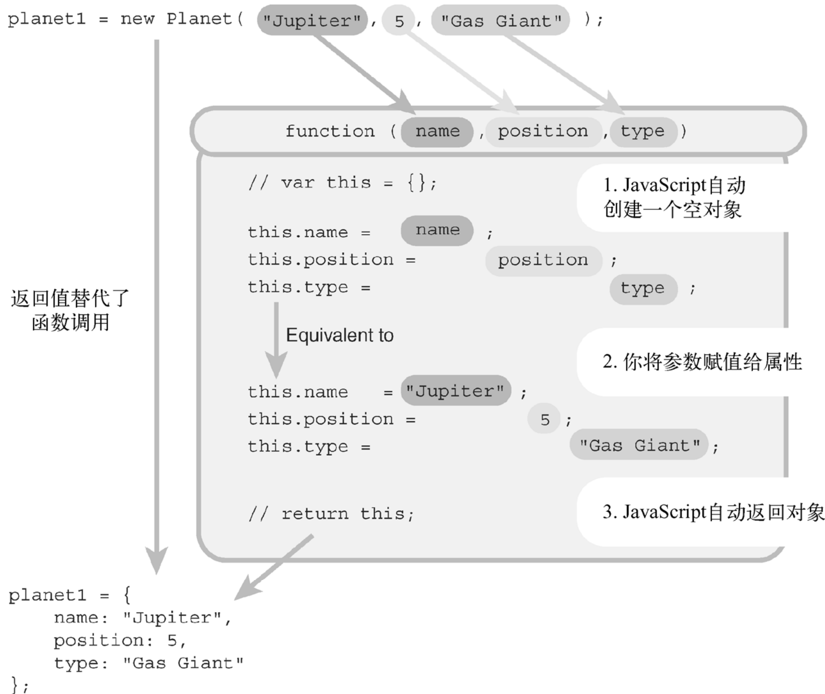


图 9-2　使用关键字 new 调用构造函数，构造函数自动创建并返回一个对象

以下代码清单 9-4 显示完整的 Planet 构造函数以及使用 new 调用 Planet。



代码清单 9-4　Planet 构造函数 (http://jsbin.com/bixico/edit?js,console)

```
var Planet = function (name, position, type){ //将函数赋值给变量，该变量的
                                              //首字母为大写字母
   this.name = name; //给this设置属性，this是JavaScript自动创建的一个空对象
   this.position = position;
   this.type = type;
   this.showPlanet = function () {
      var info = this.name + ": planet " + this.position;
      info += " - " + this.type;
      console.log(info);
   };
};
```
128

```
var planet = new Planet(      //使用 new 关键字调用该函数，告诉 JavaScript 将
                              //空对象赋值给 this
    "Jupiter",
    5,
    "Gas Giant"
    );
planet.showPlanet();
```

在以上代码中，当使用 new 调用函数时，JavaScript 才会为我们创建一个空对象，并将其分配给特殊变量 this。为了便于理解，我们可以假想 Planet 函数隐藏了第一行代码：

```
var this = {};
```

然后，我们可以为 this 设置属性，如同之前在 buildPlanet 函数中为 planet 设置属性。该函数自动返回对象并赋值给 this，因此不需要添加 return 语句。为了便于理解，我们可以假想 Planet 函数隐藏了最后一行代码：

```
return this;
```

当执行代码清单 9-4 中的代码时，新创建的对象将替换对 Planet 构造函数的调用，新对象被赋值给 planet 变量。以下代码：

```
var planet = new Planet( "Jupiter", 5, "Gas Giant" );
```

就变成：

```
var planet = {
    name: "Jupiter",
    position: 5,
    type: "Gas Giant",
    showPlanet: function () {
        var info = this.name + ": planet " + this.position;
        info += " - " + this.type;
        console.log(info);
    }
};
```

我们可以根据需要向 this 对象添加任意数量的属性和方法。以下代码清单 9-5 就对 Planet 构造函数进行了扩展，为生成的对象添加了 moons 属性和 addMoon 方法：

```
> Jupiter: planet 5 - Gas Giant
> Moons: Io, Europa.
```

129



代码清单 9-5　在 Planet 构造函数中包含一个 moons 数组 (http://jsbin.com/wiguya/edit?js,console)

```
var Planet = function (name, position, type) {
    this.name = name;
    this.position = position;
```

```
    this.type = type;
    this.moons = [];      //创建一个空数组，并将其赋值给属性 moons
    this.showPlanet = function () {
        var info = this.name + ": planet " + this.position;
        info += " - " + this.type;
        console.log(info);
        console.log("Moons:"+this.moons.join(',') + ".");  //将 moons 数组
                                                            //的元素连接形成
                                                            //一个字符串
    };
    this.addMoon = function (moon) {
        this.moons.push(moon); //使用 push 方法将新的 moon 添加到 moons 数组
    };
};
var planet = new Planet( "Jupiter", 5, "Gas Giant" );
planet.addMoon("Io");
planet.addMoon("Europa");
planet.showPlanet();
```

在第 8 章中，我们知道 push 方法用于在数组末尾添加一个新元素，join 方法用于将数组中的所有元素连接在一起形成一个字符串，并在每两个元素之间插入可选的分隔符，能否使用这些方法更新构造函数？

行星们！卫星们！勤奋无止境，让我们继续努力……

9.2.2　使用 Planet 构造函数创建一个新世界

以下代码清单 9-6 更新了 Planet 构造函数的用法。当调用 addMoon 时，可以在 moons 数组的开头添加新的卫星：unshift，这是一种我们以前从未见过的数组方法。首先，创建三个行星对象并将其赋值给变量 planet1，planet2 和 planet3，然后分别调用 showPlanet。以下只是部分输出，请注意这些卫星的顺序：

```
> Jupiter
> Planet 5 - Gas Giant
130 > Moons: Europa, Io.
```



代码清单 9-6　使用构造函数创建多个行星 (http://jsbin.com/wewewe/edit?js,console)

```
var Planet = function (name, position, type) {
    this.name = name;
    this.position = position;
    this.type = type;
    this.moons = [];
    this.showPlanet = function () {
        console.log(this.name);
        console.log("Planet " + this.position + " - " + this.type);
        console.log("Moons: " + this.moons.join(', ') + ".");
```

```
        };
        this.addMoon = function (moon) {
            this.moons.unshift(moon);    //使用 unshift 在数组开始处添加元素
        };
    };
    var planet1 = new Planet("Jupiter", 5, "Gas Giant");
    planet1.addMoon("Io");  //先添加"Io"，再添加"Europa"
    planet1.addMoon("Europa");
    var planet2 = new Planet("Neptune", 8, "Ice Giant");
    planet2.addMoon("Triton");
    var planet3 = new Planet("Mercury", 1, "Terrestrial");
    [planet1, planet2, planet3].forEach(function(planet){   //使用方括号创建
                                               //一个数组，并立即用 forEach 遍历该数组
        planet.showPlanet();
    });
```

在以上 planet1 的代码中，首先添加了 Io，然后添加了 Europa，但在输出中顺序恰恰相反。这是因为我们使用的 unshift 是在 moons 数组的开头添加元素，而不是在末尾添加元素。

9.2.3　使用 instanceof 运算符区分对象

我们使用多个不同的构造函数创建了许多不同的对象，在程序中要处理这些对象时，有时需要将一种对象类型与其他种类的对象类型区分开来，例如 item1 是一个行星对象，一个玩家对象，还是一个位置对象？JavaScript 语言中的 instanceof 运算符能够帮助解决上述问题，我们可以用它来核实一个对象是否由某个特定的构造函数所创建。假设我们已经定义了 Planet 构造函数，item1 确实是由 Planet 构造函数所创建的，因此运行以下代码片段，控制台将会显示 true。

```
var item1 = new Planet("Jupiter", 5, "Gas Giant");
console.log(item1 instanceof Planet);
```

instanceof 运算符的返回值为 true 或 false。我们将 true 和 false 称为布尔值。事实上，布尔值也仅有 true 和 false。在本书的第二部分和第三部分，读者会看到更多的 true 和 false，131
它们是决定代码运行方向的中枢，代码前进的方向取决于某些条件是否被满足。

为了更好地理解构造函数，下面我们再来看几个例子。在下一节，我们使用构造函数来处理测验中的问题和日历中的事件。

9.3　建造大师——两个构造函数的示例

一个测验中可能包括几十个问题，一个日历中可能包括成千上万的事件，而这些数量众多的问题和事件却有可能在结构上类似。因此针对这两种类型的对象貌似最适合采用构造函数来创建。

在以下代码清单 9-7 中，使用 QuizQuestion 构造函数创建了一个问题并将其显示在控制台上：

```
> What is the capital of France?
> (1) Bordeaux
> (2) F
> (3) Paris
> (4) Brussels
```



代码清单 9-7　一个测验问题的构造函数
(http://jsbin.com/vuyesi/edit?js,console)

```
var QuizQuestion = function (question, answer) {
    this.question = question;
    this.answer = answer;
    this.options = [];
    this.addOption = function (option) {
        this.options.push(option);
    };
    this.showQuestion = function (){
        console.log(this.question);
        this.options.forEach(function (option, i){  //增加第二个形参来获取
                                                    //索引实参
            console.log("(" + (i+ 1) + ") " + option);    /*使用花括号将索
                                                    *引值加 1，并将结果用在字符串中
                                                    */
        });
    };
};
var question1 = new QuizQuestion(
    "What is the capital of France?",
    "Paris"
);
question1.addOption("Bordeaux");
question1.addOption("F");
question1.addOption("Paris");
question1.addOption("Brussels");
132 question1.showQuestion();
```

在以上代码清单 9-7 中，showQuestion 函数使用 forEach 方法来遍历备选答案 options 数组。forEach 方法将每个选项及其索引值传递到显示选项及其编号的函数：

```
function (option, i) {
    console.log("(" + (i + 1) + ") " + option);
}
```

第一个选项的索引值为 0，但是我们希望显示的数字从（1）开始，因此使用（i + 1），而不是 i。将每个索引值加 1，再进行显示，如下所示：

```
> (1) Bordeaux
> (2) F
```

```
> (3) Paris
> (4) Brussels
```

以下代码清单 9-8 是一个用于创建日历中事件的构造函数。showEvent 方法生成以下输出：

```
> Annual Review
> 3/5/16, from 4.00pm to 5.00pm
```



代码清单 9-8　一个用于创建日历中事件的构造函数 (http://jsbin.com/gemiyu/edit?js,console)

```
var CalendarEvent = function (title, startDate, startTime, endTime) {
   this.title = title;
   this.startDate = startDate;
   this.startTime = startTime;
   this.endTime = endTime;
   this.showEvent = function(){    //定义一个方法来显示当前日历中事件的相关信息
      var dateString = [           //将该日期字符串的片段构建为数组中的各元素
         this.startDate,
         ", from ",
         this.startTime,
         " to ",
         this.endTime
      ].join(""); //连接数组元素以形成该日期完整的字符串
      console.log(this.title);
      console.log(dateString);
   };
};
var calEvent = new CalendarEvent(
   "Annual Review",
   "3/5/16",
   "4.00pm",
   "5.00pm"
);
calEvent.showEvent();
```
133

在以上 showEvent 方法中，我们不仅创建了一个数组，而且立即使用 join 方法形成了一个日期信息的字符串。在第 8 章中，我们介绍过 join 方法，它是一种将多个部分连接在一起生成一个完整字符串的简便方法。JavaScript 程序员过去常常使用+=，一步步地将各个子字符串连接起来，构建成为一个完整的字符串，这种方法显然费时费力。现在，我们一般不再使用+=，而是使用轻松快捷的 join 方法来连接数组元素构建字符串。下面，我们分别采用以上两种方法构建 dateString 字符串，二者的输出结果相同。

（1）使用 join 方法

```
var dateString = [
   this.startDate,
```

```
        ", from ",
        this.startTime,
        " to ",
        this.endTime
    ].join("");
```

（2）使用+ =运算符

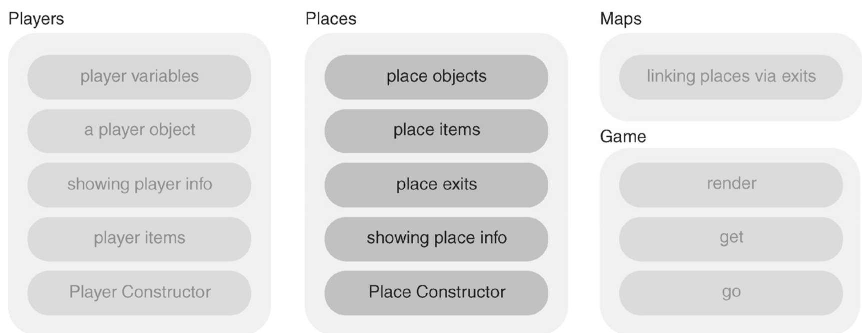
```
vardateString = this.startDate;
dateString += ", from ";
dateString += this.startTime;
dateString += " to ";
dateString += this.endTime;
```

其实，现代版本的浏览器已经很先进了，字符串连接的速度也比以前快了许多。本书在这里介绍 join 方法，以备读者将来有可能会在其他荒蛮之地遇到它。

在本章中，我们掌握了构造函数，它提供了一种标准化的流水线方法，在这条流水线上，我们使用一个模板就可以创建许多对象。每一次伟大的冒险必然会经过许许多多的地方，让我们再次探险，进入 The Crypt，使用构造函数为玩家提供足够多的掠夺地！

9.4 The Crypt——为玩家提供掠夺地

在本节中，我们将构造函数的知识应用到 The Crypt 游戏中去。图 9-3 显示通过使用构造函数来创建 Place 对象。



134 图 9-3 TheCrypt 中的游戏元素

在本书的前面章节中，我们一直专注于游戏中的玩家信息，现在开始关注探索地点。每个地点都包含以下信息：该地点的名称标题和描述、在该地点所拥有物品的集合、以及以该地点作为出口可以通往其他地点的集合。我们还需要一些方法来添加上述物品和出口，并显示信息。图 9-4 是在控制台上 Place 对象所显示的各个元素。

管理所有这些元素需要相当多的代码。我们刚刚学习的构造函数正是组织这些代码并简化多个 Place 对象创建的好方法。这将是读者目前遇到的最复杂的构造函数，所以让我们一步一步来，分阶段地构建。



图 9-4　在控制台上 Place 对象的各元素

9.4.1　构建 Place 构造函数——名称标题和描述

我们首先从一个基本的构造函数入手，用于设置 title 和 description 属性，初始输出如下所示：

```
> The Old Library
> You are in a library. Dusty books line the walls.
```



代码清单 9-9　Place 构造函数，第一部分 (http://jsbin.com/pogive/edit?js,console)

```
var Place = function(title, description){      //将构造函数赋值给以大写字母
                                                //开头的变量
   this.title = title;
   this.description = description; //为自动创建的 this 对象的属性赋值
   this.getInfo = function(){       //通过将函数赋值给属性来创建方法
      var infoString = this.title + "\n";
      infoString += this.description + "\n";
      return infoString;
   };
};
var library = new Place(            //用关键字 new 调用构造函数
   "The Old Library",
   "You are in a library. Dusty books line the walls."
);
console.log(library.getInfo());     //调用 getInfo 方法将返回值显示在控制台上
```

135

以上显示内容只是最基本的信息，并没有方框和边框。我们定义了一个 getInfo 方法，返回一个包含地点的名称标题和描述的字符串。请注意，“\n”是换行符，名称标题在一行，而描述在下一行。

现在我们已经可以创建地点了。玩家如何掠夺这些地点？我们还需要一些宝藏！

9.4.2 构建 Place 构造函数——囤积物品

当玩家探索四周环境时，他们期望拾得一些宝物，拥有这些物品他们才能披荆斩棘、战无不胜。因此，我们需要一种将物品添加到已经创建好的地点的方法，并在显示每个地点信息时显示这些物品。

以下代码清单 9-10 是对基本构造函数 9-9 的扩展，包含了物品信息，而且还使用第 7 章中的 spacer 命名空间对显示外观进行了升级，如下所示：

```
====================
=  The Old Library  =
====================
You are in a library. Dusty books line the walls.
Items:
- a rusty key
====================================
```

这看上去与图 9-4 的目标输出比较接近！在以下代码清单中并没有显示 spacer 代码，但是在 JS Bin 上有所显示。



代码清单 9-10　Place 构造函数，第二部分
(http://jsbin.com/qemica/edit?js,console)

```
var Place = function (title, description) {
    this.title = title;
    this.description = description;
136 this.items = [];                    //创建一个空数组并将其赋值给 items 属性
    this.getItems = function () {       //定义一个 getItems 方法来构建物品字符串
        var itemsString = "Items: " + spacer.newLine();
        this.items.forEach(function (item) {
            itemsString += " - " + item;
            itemsString += spacer.newLine();
        });
        return itemsString;
    };
    this.getInfo = function () {
        var infoString = spacer.box(
            this.title,
            this.title.length + 4,
            "="
        );
        infoString += this.description;
        infoString += spacer.newLine();
        infoString += this.getItems();     //在构建关于地点的 info 时，包括对
                                           //getItems 的调用
        infoString += spacer.line(40, "=") + spacer.newLine();
```

```
        returninfoString;
    };
    this.addItem = function (item) {      //定义 addItem 方法，将新物品添加到
                                          //items 数组
        this.items.push(item);
    };
};
var library = new Place(
    "The Old Library",
    "You are in a library. Dusty books line the walls."
);
library.addItem("a rusty key");    //使用 addItem 方法添加物品
console.log(library.getInfo());
```

使用 spacer 命名空间可以添加换行符，可以在名称标题周围设计一个方框，还可以使用边框结束信息字符串。关于 spacer 方法的具体用法，请参阅第 7 章相关内容。

其实我们本来可以将 items 参数作为 Place 构造函数的一部分，但是现在希望在游戏进行中能够向某个场所添加物品，因此不把 items 参数放入 Place 构造函数，使构造函数保持简单，然后再用 addItem 方法来添加物品。

那条黑暗的通道将通往何处？是龙的巢穴？是星际的桥梁？还是消失的恐龙谷？

9.4.3　构建 Place 构造函数——探索出口

如果玩家只能停留在一个场所，又怎能算是冒险呢？他们想要漫游广阔的世界。下面，我
们就来为 Place 构造函数添加最后一项内容：出口信息，即此处可以通往何处。每个场所包括 137
一个出口数组，以及一个方法，该方法用于将目的地添加到该数组。出口输出如下所示：

```
====================
=  The Old Library =
====================
You are in a library. Dusty books line the walls.
Items:
- a rusty key
Exits from The Old Library:
- The Kitchen
- The Main Hall
========================================
```

以下代码清单中再次省略了 spacer 代码。



代码清单 9-11　Place 构造函数，第三部分
(http://jsbin.com/parale/edit?js,console)

```
var Place = function (title, description) {
    var newLine = spacer.newLine(); //将 spacer.newLine（）返回的换行字符赋
                                    //值给 newLine 变量
    this.title = title;
```

```
        this.description = description;
        this.items = [];
        this.exits = [];                    //创建一个空数组，并将其赋值给 exits 属性
        this.getItems = function(){      /* 见代码清单 9-10 */    };
        this.getExits = function () {   //定义 getExits 方法来构建一个出口信息字
                                        //符串
            var exitsString = "Exits from " + this.title;
            exitsString += ":" + newLine;
            this.exits.forEach(function (exit) {
                exitsString += " - " + exit.title;
                exitsString += newLine;
            });
            return exitsString;
        };
        this.getTitle = function () {   //将名称标题的方框代码移到自己的方法代码中
            return spacer.box(
                this.title,
                this.title.length + 4,
                "="
            );
        };
        this.getInfo = function () {
            var infoString = this.getTitle(); //使用新方法来构建信息字符串
            infoString += this.description;
            infoString += newLine + newLine;
            infoString += this.getItems() + newLine;
            infoString += this.getExits();
            infoString += spacer.line(40, "=") + newLine;
            return infoString;
        };
        this.showInfo = function () {
            console.log(this.getInfo());
        };
        this.addItem = function (item) {
            this.items.push(item);
        };
        this.addExit = function (exit) {     //定义 addExit 方法，将一个新场所添
                                             //加到 exits 数组
            this.exits.push(exit);
        };
    };
    var library = new Place(
        "The Old Library",
        "You are in a library. Dusty books line the walls."
    );
    var kitchen = new Place(
```

138

```
        "The Kitchen",
        "You are in the kitchen. There is a disturbing smell."
    );
    var hall = new Place(
        "The Main Hall",
        "You are in a large hall. It is strangely empty."
    );
    library.addItem("a rusty key");
    library.addExit(kitchen);  //将 library 作为与 kitchen 和 hall 相连的接合点
    library.addExit(hall);
    library.showInfo();
```

代码貌似很长，但是相当一部分是重复前面讲过的内容。其实读者不必通读全部代码，只需每次专注于一个方法，花些时间读懂该方法即可。

首先，我们尽量采用简捷方式。spacer.newLine 方法总是返回一个新行“\n”，所以先调用它一次，并将返回值分配给 newLine 变量，以后我们就可以使用 newLine 而不是 spacer.newLine()，这样不仅可以节省一些输入，还不会影响代码的可读性。其次，我们还可以添加 getExits 和 getTitle 方法来构建场所信息字符串中的相应部分。还有，正如 addItems 方法可以用来添加物品，我们也可以采用 addExits 方法来添加出口。

exits 属性包含一个 Place 对象的数组，换句话说，每个 Place 对象都包含一个 Place 对象的集合。这种在对象中嵌套对象的能力可以为现实问题构造复杂模型。尽管一个完整的模型可能会相当复杂，但是每个组件应该相对容易理解。游戏 The Crypt 能够让我们充分享受这 139
种由简单部件构建宏伟工程的过程。让我们重温一下本章学习的内容，为玩家对象定义一个构造函数。

9.5　The Crypt——简化玩家创建代码

图 9-5 显示通过使用构造函数来创建 Player 对象。

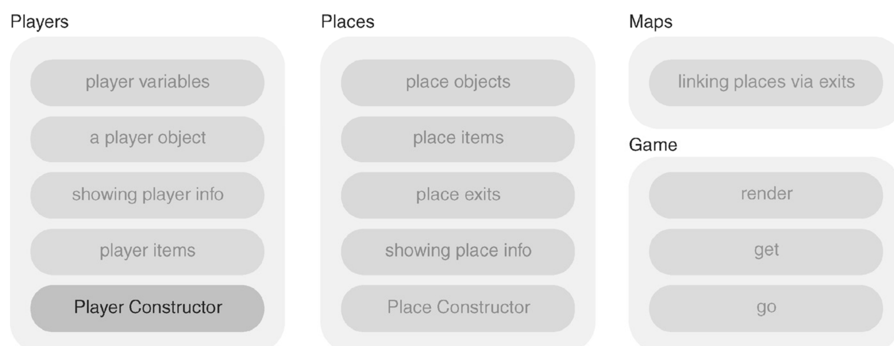


图 9-5　游戏 The Crypt 中的各元素

在第 8 章中，我们向玩家对象添加了一个物品的数组，并定义了函数，以便在显示玩家信息时能包括这些物品。但是我们依然是采用手动方式创建各个玩家，如下所示：

```
var player1 = {
   name: "Kandra",
   place: "The Dungeon of Doom",
   health: 50,
   items : ["a trusty lamp"]
};
```

我们将各个玩家显示函数分配给各个变量，变量的数量可真不少！如下所示：

```
var getPlayerName = function (player) { … };
var getPlayerHealth = function (player) { … };
var getPlayerPlace = function (player) { … };
var getPlayerItems = function (player) { … };
var getPlayerInfo = function (player, character) { … };
var showPlayerInfo = function (player, character) { … };
```

以上各个显示函数共同努力，在控制台上显示以下信息，如图 9-6 所示。

为了将物品添加给玩家，直接使用了 push 将其推入 items 数组：

140

```
player1.items.push("a rusty key");
```



图 9-6　Player 对象在控制台上所显示的元素

9.5.1　整理玩家属性

下面，我们要整合这些碎片，使用构造函数简化玩家的创建，使函数成为方法，消灭这些数量庞大的变量。Player 构造函数可以创建 Player 对象，如下所示：

```
var player1 = new Player("Kandra", 50);
```

构造函数还会添加一个方法，我们可以通过调用该方法来添加物品：

```
player1.addItem("a rusty key");
```

以下代码清单 9-12 是 Player 构造函数的代码，也使用了第 7 章 spacer 命名空间中的方法（本书代码省略了 spacer 代码，但在 JS Bin 中有所显示）。为了测试由 Player 构造函数所创建的 Player 对象，该代码清单还使用了 9.4 节中的 Place 构造函数（在本书中还是省略

了，但在 JS Bin 中有显示）。该代码清单中出现了一种新的值类型：null，在代码清单的后面会有解释。

这段代码很长（在 JS Bin 上更长），但不要担心，我们已经看过类似的代码。请读者重点查看代码注释，阅读代码清单里面的解释，并尝试在 JS Bin 上运行代码。建议读者能快速复习一下第 8 章中的 The Crypt 部分，这部分内容是以下代码的重要基础。



代码清单 9-12　Player 构造函数 (http://jsbin.com/leqahi/edit?js,console)

141

```
var Player = function (name, health) {
    var newLine = spacer.newLine();
    this.name = name;
    this.health = health;
    this.items = [];
    this.place = null;    //在创建一个场所并赋值给 place 属性之前，先将空值 null
                          //分配给 place 属性
    this.addItem = function (item) {      //定义一个方法来为物品数组添加元素
        this.items.push(item);
    };
    this.getName = function(){       //从 get 函数返回玩家信息的字符串
        return this.name;
    };
    this.getHealth = function(){
        return this.name + " has health " + this.health;
    };
    this.getPlace = function(){
        return this.name + " is in " +this.place.title;   //使用已分配给
                                                     //this.place 的 Place
                                                     //对象的 title 属性
    };
    this.getItems = function(){
        var itemsString = "Items:" + newLine;
        this.items.forEach(function(item, i){
            itemsString += " - " + item + newLine;
        });
        return itemsString;
    };
    this.getInfo = function(character){
        var place = this.getPlace();
        var health = this.getHealth();
        var longest = Math.max(place.length, health.length) + 4;
        var info = spacer.box(this.getName(), longest, character);
                                             /*使用 spacer 方法对玩家信
                                             *息进行格式处理
                                             */
        info += spacer.wrap(place, longest, character);
```

```
            info += newLine + spacer.wrap(health, longest, character);
            info += newLine + spacer.line(longest, character);
            info += newLine;
            info += " " + this.getItems();
            info += newLine;
            info += spacer.line(longest, character);
            info += newLine;
            return info;
        };
        this.showInfo = function(character){ //定义一个方法来获取玩家信息字符串并
                                              //将其显示在控制台上
        console.log(this.getInfo(character));
        };
    };
    //测试 Player 构造函数
    var library = new Place(                  //创建一个 Place 对象
        "The Old Library",
        "You are in a library. Dusty books line the walls."
142 );
    var player1 = new Player("Kandra", 50); //创建一个 Player 对象
    player1.place = library;      //将 Place 对象分配给 player
    player1.addItem("a rusty key");
    player1.addItem("The Sword of Doom");
    player1.showInfo("=");
```

这个代码清单实在是太长了！下面，我们把它拆分开来，仔细研究。

9.5.2 将函数转换为方法

在第 7 章和第 8 章中，我们定义了一些函数来构建并显示玩家信息字符串，例如：

```
var getPlayerHealth = function (player) {
    return player.name + " has health " + player.health;
};
```

我们将 Player 对象作为实参传递给函数：

```
getPlayerHealth(player1);
```

然后，该函数使用 Player 对象的属性来构建和返回信息字符串。

现在已将这些函数移动到 Player 构造函数中，并将这些函数分配给特殊对象 this 的属性：

```
this.getHealth = function () {
    return this.name + " has health " + this.health;
};
```

当我们将函数指定为对象的属性时，称之为方法。可以使用圆点运算符和圆括号来调用

方法：

```
player1.getHealth（）；
```

我们不再需要将 Player 对象作为参数传递给函数。凡是在函数体中使用“this”的地方，player1 将代替它，如下所示：

```
return this.name + " has health "+ this.health;
```

变成：

```
return player1.name +" has health "+ player1.health;
```

以上将所有函数转换为 Player 对象的方法不再需要为每个函数使用单独的变量，这样做还能够使函数定义与其操作的对象放在一起，干净利落！ 143

9.5.3　为玩家分配位置

我们需要知道玩家在游戏中的具体位置在哪里。在前面的章节中，为每个玩家的 place 属性分配了一个字符串：

```
player1.place ="The Old Library";
```

每一个位置所包含的内容不只是这个位置的名称。使用 Place 构造函数创建的 Place 对象包含名称、描述、物品数组、出口和方法。从现在开始，要把完整的 Place 对象分配给玩家：

```
var library = new Place(
   "The Old Library",
   "You are in a library. Dusty books line the walls."
);
var player1 = new Player("Kandra", 50);
player1.place = library;
```

随着玩家探索周围的环境，可以为每个新位置分配一个已经构建好的 Place 对象，来更新 place 属性。

现在，我们已将完整的 Place 对象分配给玩家的 place 属性，还需要用 getPlace 方法做一些额外的工作来构建其信息字符串：

```
this.getPlace = function（）{
   return this.name +"is in"+ this.place.title;
};
```

以前，this.place 存放玩家当前位置的名称，现在存放一个 Place 对象。我们可以通过 this.place.title 来访问地点的名称。

9.5.4　使用 null 作为对象的占位符

请注意，在代码清单 9-12 中，我们将特殊值 null 分配给构造函数中的 place 属性。这表

明我们打算在程序中使用 place 属性，但目前该属性还没有一个确定的值，只有在创建好一些位置后，才能将其分配给玩家的 place 属性。

```
player.place = null; //我们期望将来会有一个对象来此赋值
/ *其他代码* /
player.place = library; //果然有一个对象来赋值
```

定义 null 是一个独特的值类型，它既不是字符串、数字、布尔值（true 或 false）或 undefined，也不是一个对象。程序员经常使用 null，表明目前还没有分配一个对象到某变量或属性，但希望在以后会分配一个对象到该变量或属性。

现在我们已经学会采用高效的方法来创建许多对象，可以使用构造函数来构建 The
144 Crypt 中的所有地点。我们可以在各个地点添加物品，可以将各个地点连接在一起绘制地图，还可以将地点分配给玩家。目前，我们迫切需要找到一种方式，能够让玩家从一个地方移动到另一个地方。这样就真正创建了一个可以自由探索的游戏环境！

9.6 本章小结

■ 使用构造函数来创建具有类似结构的对象。

■ 将构造函数分配给以大写字母开头的变量。以这种方式来命名构造函数是一个广泛遵循的惯例：

```
var Person = function (name) {...};
```

■ 使用 new 关键字调用构造函数。将返回的对象分配给一个变量：

```
var person = new Person ("Jahver") ;
```

■ 使用特殊的 this 变量在构造函数内设置属性。this 从构造函数自动返回：

```
var Person = function (name) {
    this.name = name;
};
var person = new Person ("Jahver") ;
```

■ 像其他任何对象一样，访问返回对象的属性：

```
person.name; // Jahver
```

■ 将函数分配给属性来创建方法，使用圆点运算符和圆括号调用方法：

```
var Person = function (name) {
    this.name = name;
    this.sayHello = function () {
        return this.name +"said hi.";
    };
};
var person = new Person ("Jahver") ;
```

```
person.sayHello（）；      // Jahver says hi.
```

■ 如果希望未来分配对象给某一个变量或属性，但该对象目前还不可用，请先为该变量或属性指定值 null：

```
player1.place = null;
/ *其他代码* /
player1.place = library;
```
145

■ 使用 instanceof 运算符来检查某个对象是否由某个构造函数所创建。该运算符返回一个布尔值，true 或 false：

```
var person = new Person（"Jahver"）；
person instanceof  Person;         // true
person instanceof  Planet;         // false
```
146

第 10 章　方括号运算符：灵活的属性名称

本章内容包括：

- 方括号运算符替代圆点运算符
- 使用方括号运算符设置和获取属性
- 方括号运算符的灵活性
- 如何在 The Crypt 中创建一个工作版游戏

在第 3 章中，我们学习了如何使用花括号创建对象，并使用圆点运算符获取和设置属性。我们学会使用对象来创建玩家、地点、行星、博客帖子，以及测验和日程事件，还能够通过添加函数作为属性来创建方法，并将对象作为参数和返回值传递给函数。对象简直就是整个 JavaScript 世界的核心！

在本章中，读者将了解使用对象属性的一种新方法，它能够让我们更灵活地使用属性名称，使用变量作为键，并拥有在程序运行中根据数据生成新属性的能力。

作为本书第一部分的谢幕作品，本章将完成 The Crypt 的工作版本，最终给玩家提供探
147 索地图和收集宝贝的机会！方括号运算符在游戏改进过程中发挥了以下作用：管理游戏中各位置之间的链接，创建与网络链接的 Place 对象，为冒险活动增加神秘感。

10.1　用方括号运算符替代圆点运算符

在前几章中，我们已经学会使用圆点运算符设置和获取对象属性。

```
question1.question = "What is the capital of France?";
console.log(player1.name);
this.title = title;
```

属性名称（键）通过圆点符号与变量名连接在一起，这样就可以访问变量的属性。除此之外，JavaScript 还提供了另外一种访问属性的方法：通过将属性的键作为字符串包含在方括号之间来设置和获取属性，如下所示：

```
question1["question"] = "What is the capital of France?";
console.log(player1["name"]);
this["title"] = title;
```

这种新方法为我们把字符串作为键以及在程序运行时添加动态属性提供了更大的灵活性。例如，我们要设置一个 states 对象，把美国州名缩写作为值，把各州的全名作为键，我们完全可以使用圆点运算符来获取 Ohio（俄亥俄州）的缩写，如下所示：

```
console.log(states.ohio); // OH
```

但是，因为某些州名中有个讨厌的空格，例如 New Hampshire，就无法再使用圆点运算符来输出，如下所示：

```
console.log(states.new hampshire); // 错误!
```

使用方括号运算符可以解决上述问题（图 10-1）：

```
console.log(states["new hampshire"]); // NH
```

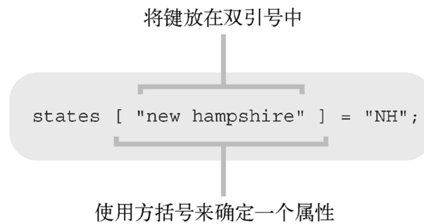


图 10-1　使用方括号运算符设置属性

148

假设我们有一个使用 states 对象的 getStateCode 函数，方括号运算符可以使用函数的参数作为键：

```
var getStateCode = function (stateName) {
   var stateCode = states[stateName];
   return stateCode;
};
```

如果使用一个函数参数作为键，就不能使用圆点运算符。表 10-1 列出了需要使用方括号运算符的多种情况。

表 10-1　各种键的执行情况

键的情况	语句尝试	执行结果
在对象字面量中使用 ohio 作为键	states = { ohio : "OH" };	成功
在对象字面量中使用 new hampshire 作为键	states = { new hampshire : "NH" };	失败
在对象字面量中使用"new hampshire"作为键	states = { "new hampshire" : "NH" };	成功
用圆点运算符访问键 maryland	states.maryland = "MD";	成功
用圆点运算符访问键 southcarolina	states.south carolina = "SC";	失败
用方括号运算符访问键 south carolina	states["south carolina"] = "SC";	成功
函数参数作为键，用圆点运算符访问	function (stateName) { return states.stateName; }	失败
函数参数作为键，用方括号运算符访问	function (stateName) { return states[stateName]; }	成功

以下几个示例将有助于读者理解上述概念。 149

10.1.1　使用方括号——人的姓名作为键

假设我们需要保存一个简单的年龄记录，可以使用 ages 对象，每个人的姓名是键，其

年龄是相应的值。代码清单 10-1,10-2 和 10-3 采用以上方法，在控制台上输出两人的年龄：

```
> 56
> 21
```

在以下第一个代码清单中，使用方括号运算符设置属性，并使用圆点运算符访问属性。



代码清单 10-1　方括号运算符用于对象的属性
(http://jsbin.com/kipedu/edit?js,console)

```
var ages = {};
ages["Kandra"] = 56;          //用名字作为键，用方括号运算符设置属性
ages["Dax"] = 21;
console.log(ages.Kandra);     //用名字作为键，用圆点运算符获得属性
console.log(ages.Dax);
```

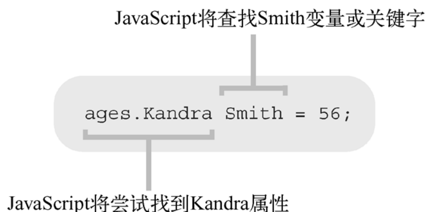
这两种方法对于代码清单 10-1 中使用的键是等效的。但是，如果在属性名称中包含空格，则必须使用方括号。下面的代码清单是一个类似的示例，但姓名部分使用了全名。



代码清单 10-2　更长的字符串作为键
(http://jsbin.com/toviya/edit?js,console)

```
var ages = {};
ages["Kandra Smith"] = 56;
ages["Dax Aniaku"] = 21;
console.log(ages["Kandra Smith"]);
console.log(ages["Dax Aniaku"]);
```

使用圆点运算符访问带有空格的属性名称就会给 JavaScript 带来混乱。JavaScript 将 ages.Kandra Smith 解释为 ages.Kandra，而 Smith 会被看作一个单独的变量名称，如图 10-2 所示。



150

图 10-2　圆点运算符不适用于属性名称中带有空格的情况

方括号运算符可以解决上述问题，能够更灵活地使用属性的名称。尤其是当程序员事先不知道要添加属性的键的信息时，方括号允许在代码运行中添加属性，这些属性可能来自用户的输入或者程序从文件或数据库中获取的信息。例如，在下一个代码清单中，读者将看到一个 addAge 函数，用于向 age 对象添加新的人员信息。运行此程序代码，就能够在控制台上使用该函数添加新人。



代码清单 10-3　使用函数添加年龄 (http://jsbin.com/pipuva/edit?js,console)

```
var ages = {};
var addAge = function (name, age) {      //包含一个 name 形参，以后可以将一个
                                         //名称作为实参传递给该函数
    ages[name] = age;
};
addAge("Kandra Smith", 56);              //将传递给函数的名称作为键
addAge("Dax Aniaku", 21);
console.log(ages["Kandra Smith"]);
console.log(ages["Dax Aniaku"]);
```

在代码清单 10-3 中，使用 addAge 函数中的 name 参数作为 ages 对象的键。当调用 addAge 时，第一个实参将分配给形参 name。如下所示：

```
addAge("Kandra Smith", 56);
```

对于函数调用中的两个实参，"Kandra Smith"分配给 name，56 分配给 age，因此

```
ages[name] = age;
```

变成

```
ages["Kandra Smith"] = 56
```

这种在方括号内使用变量来设置属性的能力使我们能够在程序中动态创建和改变对象，更进一步使用 10.1.2 和 10.2 节中的技巧。

如果要创建具有已存在属性的对象，请使用方括号，并用逗号分隔键-值对。如果使用引号将键引起来，键中可以包含空格。下一个代码清单采用了以上做法，还介绍了 Object.keys 方法，该方法返回一个包含对象中所有键的数组。该代码清单中，把 keys 数组输出到控制台，如下所示：

```
> ["Kandra Smith", "Dax Aniaku", "Blinky"]
```
151



代码清单 10-4　使用 Object.keys (http://jsbin.com/mehuno/edit?js,console)

```
var ages = {  // Blinky 不需要使用引号，其他两个需要使用引号才能保证键中的空格有效
    "Kandra Smith" : 56,
    "Dax Aniaku" : 21,
    "Blinky" : 36
};
var keys = Object.keys(ages); 使用 Object.keys 获取设置在 ages 对象属性上的名称数组
console.log(keys);
```

Object 是一个内置的 JavaScript 对象，它提供了许多方法，其中之一称为 keys。因为 Object.keys 返回一个数组，所以可以使用 forEach 将每个键传递给一个函数。代码清单 10-5

中将每个键依次传递给 console.log 函数，并将键输出到控制台，如下所示：

```
>Kandra Smith
>Dax Aniaku
>Blinky
```



代码清单 10-5　使用 forEach 在 Object.keys 上进行遍历 (http://jsbin.com/seteco/edit?js,console)

```
var ages = {  //获取与 ages 对象属性上的名称数组
    "Kandra Smith" : 56,
    "Dax Aniaku" : 21,
    "Blinky" : 36
};
var keys = Object.keys(ages);
keys.forEach(function (key) { //将每个属性上的名称输出到控制台上
    console.log(key);
});
```

关于使用方括号运算符包含复杂的属性名称，请读者再看一个示例。

10.1.2　最大限度地利用方括号运算符——单词统计

作为单位的社交媒体专家，你的任务是分析推特。首要任务就是统计一批推文中每个单词的使用次数。图 10-3 显示了一个推特分析程序在控制台上输出的部分内容，程序在代码清单 10-6 中显示，程序中使用 words 对象来记录字数。为了节省空间，只选用三条
152 推文。

```
#ESERO: 1,
#pet: 1,
#STEMeducation: 1,
&: 1,
-: 1,
a: 3,
A: 1,
about: 1,
across: 1,
age: 1,
is: 4,
is…a: 1,
it: 3,
It: 1,
just: 1,
learn: 1,
LIVE: 1,
look: 1,
loyalty: 1,
M:I5: 1,
subscribers!: 1,
sure,: 1,
technology: 1,
test: 1,
that: 2,
the: 4,
This: 1,
time: 1,
to: 8,
Trying: 1,
```

图 10-3　统计推文中的单词



代码清单 10-6　统计推文中的单词使用次数 (http://jsbin.com/figati/edit?js,console)

```
var tweets = [
    "Education is showing business the way by using technology to share
    ➥ information. How do we do so safely?",
    "Enjoy a free muffin & coffee with Post Plus, our new loyalty club
    ➥ exclusive to subscribers!",
    "We're LIVE on Periscope right now answering all your #pet questions
    ➥ - tweet us yours now!"
```

```
];
var words = {};
var tweetText = tweets.join(" ");          //将所有的推文连接形成一个长字符串
var tweetWords = tweetText.split(" ");     //使用 split 方法创建一个单词数组
tweetWords.forEach(function (word) {       //为每个单词创建一个属性，将其值设置为 0
   words[word] = 0;
});
tweetWords.forEach(function (word) {       //单词每出现一次，就将其值增加 1
   words[word] = words[word] + 1;
});
console.log(words);
```

该程序在短短几行代码中完成了多项工作。首先，使用 join 数组方法（在第 8 章中介绍过）连接所有的推文形成一个长字符串，在每对推文之间有一个空格。然后，使用 split 方法创建一个存放所有单词的数组。split 是一个内置的字符串方法，可以用来将一个字符串拆分为多个部分，每个部分作为数组的一个元素。例如，为 message 变量分配一个字符串，如下所示：

```
var message = "I love donuts";
```

可以通过调用该变量的 split 方法将字符串拆分为三个元素，并将它们存放在数组中，如下所示：

```
console.log(message.split(" "));
> ["I", "love", "donuts"]
```
153

传递给 split 方法的参数是该函数决定用来分割文本的分隔符字符串。上一个示例中使用空格作为分隔符，其实任何字符串都是可以用作分隔符的。以下示例中使用逗号作为分隔符：

```
var csv = "Kandra Smith,50,The Dungeon of Doom";
var details = csv.split(",");
console.log(details);
> ["Kandra Smith", "50", "The Dungeon of Doom"]
```

如果把空字符串作为参数传递给 split，它将创建一个含有文本中所有单个字符的数组：

```
var message = "I love donuts";
console.log(message.split(""));
> ["I", " ", "l", "o", "v", "e", " ", "d", "o", "n", "u", "t", "s"]
```

了解 split 方法之后，我们返回上述推特分析器程序。一旦生成了分割好的单词数组，代码清单 10-6 遍历该单词数组两次。第一次遍历会创建以下属性：单词为键、零为值，例如第一次遍历["I", "love", "donuts"]会创建以下属性：

```
words["I"] = 0;
words["love"] = 0;
words["donuts"] = 0;
```

如果一个单词出现多次，其属性每次将被重置——稍显冗余但不是问题。第二次遍历，对每次出现的单词，代码都会将其属性值加 1。

```
words["I"] = words["I"] + 1;
words["love"] = words["love"] + 1;
words["donuts"] = words["donuts"] + 1;
```

或者，我们可以使用+=运算符为每个单词的计数增加 1。我们曾经学过+=用于连接字符串，现在学习+=用于数值的计算。+=可以为已经存在的数值累加一个新数值。以下两条语句是等效的：

```
words[word] = words[word] + 1;
words[word] += 1;
```

事实上，如果只是想每次加 1，++运算符就可以实现。以下两条语句也是等效的：

```
words[word] += 1;
words[word]++;
```

因为我们想让代码更加易于理解，如果使用++就显得有些过于精简了（++还有一些其他的复杂含义，本书不再赘述）。读者可能会在程序中偶然碰到++，但在本书的第四部分之前不会再使用++。

154

所有的单词已经遍历了两次，在第二次遍历结束之后，单词的每个属性都可以作为单词的键，单词的统计数值作为值。请在 JS Bin 上运行程序示例，并添加更多的推文或其他来源的文本进行分析。

通过简单调整，我们就可以对字母进行统计，而不再仅限于单词统计。以下代码清单 10-7 显示了具体的调整方法。



代码清单 10-7　统计推文中的字母
(http://jsbin.com/rusufi/edit?js,console)

```
var tweets = [ /* 与代码清单 10-6 相同 */ ];
var letters = {};
vartweetText = tweets.join("");
vartweetLetters = tweetText.split("");  //将空字符串传递给 split 来创建单个
                                         //字符的数组
tweetLetters.forEach(function (letter) {
   letters[letter.toLowerCase()] = 0;   //将所有字母转换为小写，并将其作为键
});
tweetLetters.forEach(function (letter) {
   letters[letter.toLowerCase()]+=1; //使用+=运算符为每个字母的出现次数累加 1
});
console.log(letters);
```

以上代码清单的工作原理与代码清单 10-6 相同，都是采用 join 和 split 创建一个数组，然后两次遍历数组进行统计。

在下一节中，我们将使用方括号运算符来处理任意字符串，以便在 The Crypt 中对出口进行管理。

10.2　The Crypt——使游戏出口更加刺激好玩

本节中，我们把方括号运算符的知识应用到游戏 The Crypt 中。图 10-4 显示了本节的重点：通过出口连接其他各场所。

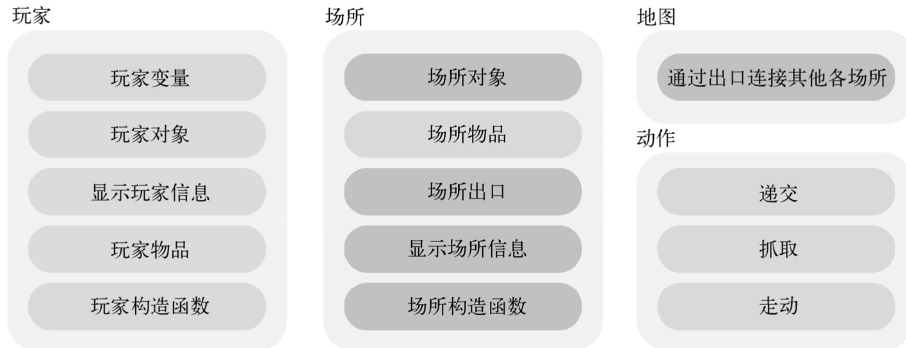


图 10-4　The Crypt 中的游戏元素 155

目前看来，我们在第 9 章中定义的 Place 构造函数对游戏当前出口暴露的信息量有点太多了：

```
    ================
    =  The Kitchen  =
    ================
    You  are  in  a  kitchen.  There  is  a
disturbing smell.
    Items:
    - a piece of cheese
    Exits from The Kitchen:
    - The Kitchen Garden
    - The Kitchen Cupboard
    - The Old Library
    ============================
```

厨房
你在厨房里,这里有一种令人不舒服的气味。
物品:
-一块奶酪
从厨房的出口可以通往：
-厨房花园
-厨房橱柜
-旧书房

以上列出了当玩家离开当前位置时会看到的内容。在本节中，我们只明确出口通往的方向而并不明确出口通往的目的地，从而为游戏增加一些神秘感，如下所示：

```
    Exits from The Kitchen:
    - west
    - east
    - south
```

厨房的出口通往：
-西边
-东边
-南边

这样，玩家只能先朝着某个方向前进，在到达那里之后才能知道附近有什么，这样就增

加了一点悬念。

为了帮助读者专注于使用带有方括号运算符的对象来优化出口（这毕竟是本章的主要内容），我们将创建一个简化版 Place 构造函数，并对其进行实验。然后，在 10.2.4 节中，再使用方括号运算符的新功能来更新第 9 章中的构造函数。

10.2.1 使用对象存放出口

为了增加这一层额外的神秘感，我们使用一个 exits 对象而不是 exits 数组。对象的键是方向，诸如"north"或"the trapdoor"，对象的值是目的地。代码清单 10-8 在控制台上显示了方向和目的地，如下所示。稍后，我们将把目的地隐藏起来。

```
>north goes to The Kitchen
>the trapdoor goes to The Dungeon
```



代码清单 10-8　一个 exits 出口对象
(http://jsbin.com/daqato/edit?js,console)

```
var Place = function (title) {     //定义一个简单的 Place 构造函数
   this.title = title;
};
var kitchen = new Place("The Kitchen"); //使用 Place 构造函数创建两个 Place
                                        // 对象
156 var dungeon = new Place("The Dungeon");
var exits = {};    //创建一个空对象并将其分配给 exits 变量
exits["north"] = kitchen;//将 Place 对象分配给 exits 对象的属性，方向作为键
exits["the trapdoor"] = dungeon;
var keys = Object.keys(exits);
keys.forEach(function (key) {      //使用每个 Place 的标题属性显示目的地
   console.log(key + " goes to " + exits[key].title);
});
```

在以上代码清单 10-8 中，首先定义了一个非常简单的 Place 构造函数。我们记得在第 9 章中，当使用 new 关键字调用构造函数时，JavaScript 会自动创建一个空对象并将其分配给特殊的 this 变量。我们将 this 的 title 属性设置为 title 参数的值，如下所示：

```
var Place = function (title) {
   this.title = title;
};
```

立即使用这个构造函数来创建两个 Place 对象：

```
var kitchen = new Place("The Kitchen");
var dungeon = new Place("The Dungeon");
```

简洁起见，使用一个单独的 exits 变量，随后，让它成为 Place 构造函数的一部分。这样，就创建了一个空对象并将其分配给 exits 变量：

```
var exits = {};
```

现在我们拥有了两个 Place 对象：kitchen 和 dungeon，将把它们设置为不同方向的目的地。我们在 exits 对象上创建相应的属性：方向为键，目的地为值，如下所示：

```
exits["north"] = kitchen;
exits["the trapdoor"] = dungeon;
```

最后，使用 forEach 方法将每个键（换句话说，就是每个方向）依次传递给指定的函数：

```
console.log(key + " goes to " + exits[key].title);
```

因为键是"north" 和 "thetrapdoor"，所以该语句等价于下面的代码：

```
console.log("north" + " goes to " + exits["north"].title);
console.log("the trapdoor" + " goes to " + exits["the trapdoor"].title);
```
157

又因为 exits["north"]是 kitchen 对象，exits["thetrapdoor"]是 dungeon 对象，所以上述代码就变成：

```
console.log("north" + " goes to " + kitchen.title);
console.log("the trapdoor" + " goes to " + dungeon.title);
```

以上代码的输出结果都与代码清单 10-8 前面的两行输出内容相同。

10.2.2　创建一个添加并显示出口的函数

我们使用方括号运算符成功地关联了方向和目的地。在以下代码清单 10-9 中，添加了两个辅助函数：addExit 和 showExits，用于简化对出口的添加和显示，其输出结果与代码清单 10-8 相同：

```
>north goes to The Kitchen
>the trapdoor goes to The Dungeon
```



代码清单 10-9　添加并显示出口的函数
(http://jsbin.com/mibube/edit?js,console)

```
var Place = function (title) {
   this.title = title;
};
var kitchen = new Place("The Kitchen");
var dungeon = new Place("The Dungeon");
var exits = {};
var addExit = function (direction, place) {   //定义 addExit 函数为指定方向
                                              //添加一个场所
   exits[direction] = place;
};
var showExits = function () {       //定义 showExit 函数为每个出口显示目的地
   var keys = Object.keys(exits);
   keys.forEach(function (key) {
```

```
        console.log(key + " goes to " + exits[key].title);
    });
};
addExit("north", kitchen);    //两次调用 addExit，添加两个出口
addExit("the trapdoor", dungeon);
showExits();                  //调用 showExits 显示所有出口的相关信息
```

158

以上代码中，addExit 函数接受两个参数：表示方向的 direction 字符串和表示场所的 Place 对象。direction 字符串成为出口 exits 对象上一个新的键，Place 对象变为相应的值。如下所示：

```
addExit("north", kitchen);
```

将执行下面的代码

```
exits["north"] = kitchen;
```

showExits 函数遍历 exits 对象所有的键（即遍历了所有的方向），并显示每个方向的目的地。

10.2.3 设置每个场所对象的出口集合

我们已经了解如何使用 exits 对象来为方向和目的地建模，但每个场所对象都需要一个自己的出口集合，你一定不想把仙女游乐园的出口与末日地牢的出口混在一起。在代码清单 10-10 中，将 exits 对象移到了 Place 构造函数中。为了测试新的 Place 构造函数的功能，需要创建三个场所：书房（library）、厨房（kitchen）和花园（garden），并为厨房添加两个出口，如下所示：

```
> Exits from The Kitchen:
>south
>west
```



代码清单 10-10　Place 构造函数中的出口对象 (http://jsbin.com/foboka/edit?js,console)

```
var Place = function (title, description) {
    this.title = title;
    this.exits = {};        //在构造函数中添加一个 exits 对象，以便创建的每个场所都
                            //会获得一组出口
    this.addExit = function (direction, exit){    //将 addExit 函数设置
                                                  //为每个 Place 对象的方法
        this.exits[direction] = exit;
    };
    this.showExits = function () {  //将 showExits 函数设置为每个 Place 对象
                                    //的方法
        console.log("Exits from " + this.title + ":");
        Object.keys(this.exits).forEach(function (key) {
            console.log(key);
```

```
        });
    };
};
var library = new Place("The Old Library");  //使用 Place 构造函数创建
                                             //Place 对象
var kitchen = new Place("The Kitchen");
var garden = new Place("The Kitchen Garden");
kitchen.addExit("south", library); //使用圆点运算符调用将厨房连接到书房和花园
                                   //的 addExit 方法
kitchen.addExit("west", garden);
kitchen.showExits();          //使用圆点运算符调用 showExits 方法
```

159

在以上代码中，addExit 和 showExits 函数与 Place 和 Exit 对象一起使用，我们将这两个函数与构造函数中的其他场所代码捆绑在一起，意义在于之后就可以使用圆点运算符和括号来调用这些函数：

```
kitchen.addExit("south", library);
kitchen.showExits();
```

以下代码清单 10-11 使用相同的 Place 代码构建一个更大的地图，将四个地方连接在一起，如图 10-5 所示。



图 10-5　拥有四个场所位置的地图

其中两个场所位置的输出结果显示在控制台上：

```
> Exits from The Old Library:
>north
> Exits from The Kitchen:
>south
>west
>east
```



代码清单 10-11　拥有四个场所位置的地图
(http://jsbin.com/bufico/edit?js,console)

```
var Place = function (title, description) {   //使用与代码清单 10-10 相同的
                                               //Place 构造函数
    this.title = title;
    this.exits = {};
    this.addExit = function (direction, exit) {
        this.exits[direction] = exit;
    };
    this.showExits = function () {
```

```
            console.log("Exits from " + this.title + ":");
            Object.keys(this.exits).forEach(function (key) {
                console.log(key);
            });
        };
    };
    var library = new Place("The Old Library");   //使用构造函数创建四个 Place
                                                   //对象
    var kitchen = new Place("The Kitchen");
    var garden = new Place("The Kitchen Garden");
160 var cupboard = new Place("The Kitchen Cupboard");
    library.addExit("north", kitchen); //添加通向厨房的出口
    garden.addExit("east", kitchen);
    cupboard.addExit("west", kitchen);
    kitchen.addExit("south", library); //添加通往书房的出口
    kitchen.addExit("west", garden);
    kitchen.addExit("east", cupboard);
    library.showExits();               //显示书房和厨房的出口
    kitchen.showExits();
```

我们注意到，一旦定义了一个 Place 构造函数，就可以根据需要用它创建和连接许多场所。我们可以将 Place 构造函数从一个冒险场所移植到另一个场所，构造函数本身的代码保持不变，需要改变的只是我们创建的地点，即地图数据。

10.2.4 将 exits 对象添加到完整的 Place 构造函数

在 10.2 节中，我们已经学会如何将 exits 对象与方括号运算符结合起来管理游戏中 Place 对象之间的连接。为了重点学习方括号，我们使用了简单版 Place 构造函数来创建代码。现在我们将增强版的出口与第 9 章中的构造函数结合起来，为图 10-6 所示的每个地方生成格式化输出，如下所示：

```
===============
= The Kitchen =
===============
You are in a kitchen. There is a disturbing smell.

Items:
   - a piece of cheese

Exits from The Kitchen:
   - south
   - west          <-------------  各方向的列表
   - east
========================================
```

图 10-6　调用 showInfo 时 Place 对象在控制台上的输出

以下代码清单 10-12 显示了完整的 Place 构造函数代码。大部分代码都在第 9 章中讨论过，如果需要，可以温习这些知识。新的出口代码以粗体显示并添加了注释。JS Bin 上的代
161 码清单包括了在下一节中讨论的地图信息。



代码清单 10-12　Place 构造函数
(http://jsbin.com/zozule/edit?js,console)

```
var Place = function (title, description) {
   var newLine = spacer.newLine();
   this.title = title;
   this.description = description;
   this.items = [];
   this.exits = {};      //将 exits 属性从数组更改为对象
   this.getItems = function () {
      var itemsString = "Items: " + newLine;
      this.items.forEach(function (item) {
         itemsString += " - " + item;
         itemsString += newLine;
      });
      return itemsString;
   };
   this.getExits = function () {
      var exitsString = "Exits from " + this.title;
      exitsString += ":" + newLine;
      Object.keys(this.exits).forEach(function (key) {  //更新 getExits 方
                                                          //法来显示方向而不
                                                          //显示目的地
         exitsString += " - " + key;
         exitsString += newLine;
      });
      return exitsString;
   };
   this.getTitle = function () {
      return spacer.box(
         this.title,
         this.title.length + 4,
         "="
         );
   };
   this.getInfo = function () {
      var infoString = this.getTitle();
      infoString += this.description;
      infoString += newLine + newLine;
      infoString += this.getItems() + newLine;
      infoString += this.getExits();
      infoString += spacer.line(40, "=") + newLine;
      return infoString;
   };
   this.showInfo = function () {
      console.log(this.getInfo());
   };
   this.addItem = function (item) {
```

```
        this.items.push(item);
    };
    this.addExit = function (direction, exit) {     //更新 addExit 函数以
                                                    //使用 exits 对象
        this.exits[direction] = exit;
    };
};
```

162

使用这个最新版本的 Place 构造函数，可以创建场所对象，管理这些场所的物品和出口，并在控制台上显示格式化的信息（由 spacer 提供）。

10.2.5 测试 Place 构造函数

为了测试 Place 构造函数，我们对代码清单 10-11 中的地图进行了重建，现在地图上连接了四个地方：kitchen、library、garden 和 cupboard。以下代码清单 10-13 显示了创建地图的代码，输出结果如图 10-6 所示。



代码清单 10-13 测试 Place 构造函数
(http://jsbin.com/zozule/edit?js,console)

```
var kitchen = new Place(  //使用 new 关键字调用 Place 构造函数以创建场所对象
    "The Kitchen",
    "You are in a kitchen. There is a disturbing smell."
);
var library = new Place(
    "The Old Library",
    "You are in a library. Dusty books line the walls."
);
var garden = new Place(
    "The Kitchen Garden",
    "You are in a small, walled garden."
);
var cupboard = new Place(
    "The Kitchen Cupboard",
    "You are in a cupboard. It's surprisingly roomy."
);
kitchen.addItem("a piece of cheese");   //为每个场所添加物品
library.addItem("a rusty key");
cupboard.addItem("a tin of spam");
kitchen.addExit("south", library);      //为厨房添加出口
kitchen.addExit("west", garden);
kitchen.addExit("east", cupboard);
library.addExit("north", kitchen);      //添加通往厨房的出口
garden.addExit("east", kitchen);
cupboard.addExit("west", kitchen);
kitchen.showInfo();     //在控制台上显示厨房的信息
```

在创建地图代码之后，通过调用 kitchen.showInfo 来测试 kitchen 对象及其物品和出口是否已按预期创建。 163

10.3　The Crypt——开始游戏！

即将大功告成！为了给玩家建立并展示一片探索天地，我们已经拥有：

- spacer 命名空间
- Player 构造函数
- Place 构造函数
- 地图创建代码

在玩家能够自由冒险之前，我们还需要添加一项内容：

- 游戏的控制部分

图 10-7 显示了在本节中将要创建的三个游戏控制功能。

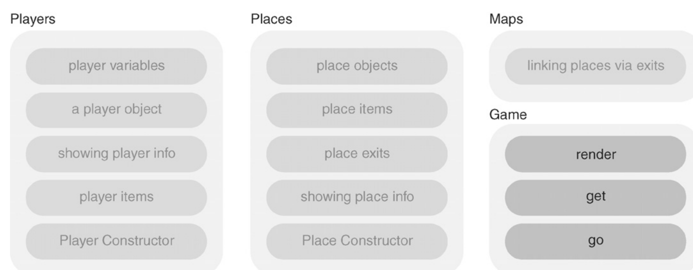


图 10-7　The Crypt 中的游戏元素

玩家能够在控制台提示符下发布命令，从一个地方移动到另一个地方，并捡拾他们找到的物品。例如，玩家要移动到北边并拿起一个物品，就在控制台提示符下输入：

```
>go("north")
>get()
```

因为使用了构造函数，游戏控制代码实际上很短小，如下所示。



代码清单 10-14　运行游戏
(http://jsbin.com/sezayo/edit?js,console)

```
var render = function () {    //定义 render 函数来清除控制台、玩家和场所信息
    console.clear();
    player.place.showInfo();
    player.showInfo("*");
};
var go = function (direction) {    //定义 go 函数，将玩家移动到指定方向的地方
    player.place = player.place.exits[direction];
    render();
    return "";
};
var get = function () {       //定义 get 函数，让玩家捡拾物品
```

164

```
        var item = player.place.items.pop();
        player.addItem(item);
        render();
        return "";
    };
    var player = new Player("Kandra", 50);  //创建玩家，给他们一把剑，并把他们放
                                            //在厨房里
    player.addItem("The Sword of Doom");
    player.place = kitchen;
    render();     //调用 render 来显示玩家和场所的初始信息
```

在以上代码中，创建了三个函数：render、go 和 get，下一节将详细讲解每个函数的功能。go 和 get 结束时将返回一个空字符串""。在控制台上调用这些函数，会自动显示其返回值。返回一个空字符串的意义在于防止控制台上显示 undefined。

10.3.1　刷新显示——render

在每个游戏开始和每次玩家采取行动时，我们都希望刷新控制台上的显示信息。Place 对象和 Player 对象都采用 showInfo 方法来显示它们当前的状态，例如：

```
player.showInfo("*");
kitchen.showInfo();
```

程序将玩家的当前位置分配给 place 属性，显示当前位置的信息需要使用以下代码：

```
player.place.showInfo();
```

我们可以将程序显示的每一条文本信息源源不断地添加并保留在控制台上，但这样显得很冗长。如果在玩家每一次行动之后都清除控制台上的所有信息，就可以让玩家在下次行动时有一个干净空白的控制台界面，这样会让人感觉更清爽。可以采用 clear 方法来清除控制台上的文本，如下所示：

```
console.clear();
```

玩家在移动和捡拾物品后刷新控制台，这时并不需要重复显示代码。我们将显示代码包
165 装在 render 函数内就可以根据需要进行调用。

10.3.2　探索地图——go

玩家想要移动到新位置，只要按照指定方向调用 go 函数即可：

```
go("south");
```

找到玩家指定方向的目的地，就为其赋值：

```
player.place = player.place.exits[direction];
```

在赋值运算符右侧的表达式中，使用 exits[direction] 指定方向，使用方括号运算符获取指定方向的 Place 对象，使用圆点运算符访问 exits 对象：

```
player;                     //玩家对象
player.place;               //玩家的当前位置
                            //Place 对象
player.place.exits;         //当前位置的出口
                            //对象的键为方向
                            //值为目的地
player.place.exits[direction];     //指定方向的场所
                                   //Place 对象.
```

因为玩家的位置已经有所移动，因此将目的地的 Place 对象分配给玩家的 place 属性，替换他们的旧位置。

10.3.3　收集物品——get

玩家通过调用 get 函数来捡拾物品：

```
get();
```

玩家从当前位置拿走了物品，并将其添加到玩家的物品数组中。当前位置是 player.place，当前位置的物品保存在 player.place.items 数组中，要移除并返回 items 数组中的最后一个物品，需要使用 pop 方法：

```
var item = player.place.items.pop();
```

然后将该物品添加到玩家的物品集合中：

```
player.addItem(item);
```

就是这样，玩家拥有了移动的能力，可以从一个地方勇敢地移动到另一个地方，并收获他们找到的宝藏。 166

10.3.4　设计一个更大的冒险游戏——Jahver 的船

我们所期待的冒险世界绝对不仅仅是围绕厨房的 4 个位置。想要涉足更神秘的领域，再造反乌托邦的未来，编织充满阴谋的密网，我们只需改变代码的地图部分：创建地点，通过出口把这些地点连接起来，再给这些地点存放一些有趣的器物。

在开始之前，在 JS Bin 上试试另一个简单的冒险游戏，http://jsbin.com/yadabo/edit?console。只要单击 Run 就可以启动游戏，该游戏发生在一个名为 The Sparrow（麻雀）的小货轮上。目前，你已经拥有物品 blaster，这里还有 6 个其他物品等着你去搜寻。打开 JavaScript 面板，然后尝试添加更多的场所，祝你搜寻愉快！

10.4　下一步目标

现在，我们已经拥有这样一款工作版游戏，但 The Crypt 的代码还不是很强大，仍然存在许多问题。例如，玩家如果输入一个尚不存在的方向，就会轻易导致游戏崩溃。玩家可以在控制台上访问所有游戏对象，这也会造成大问题。例如，玩家可以轻而易举地授予自己额

外的宝藏，轻而易举就能到达一个新的位置，如下所示：

```
>player.addItem("riches beyond my wildest dreams")
>player.place = treasureRoom
```

游戏怎能如此一蹴而就？游戏中必须有更多的挑战，有些难题只能由特定的物品才能破解！

在本书的第二部分将解决上述问题，具体方法如下：将代码拆分为模块、阻止访问对象、设置代码运行时的条件、将数据与数据显示分离。随着程序变得日趋庞大和复杂，这样的代码组织结构将帮助开发者更好地设计和管理程序，而且还为程序将来用于网络环境（本书第三部分）未雨绸缪。

10.5 本章小结

- 使用方括号之间的字符串来指定属性名称：

```
player["Kandra"] = 56;
exits["north"] = kitchen;
words["to"] = words["to"] + 1;
```

- 特别注意，当属性名称含有空格符或其他字符时，请使用方括号：

```
player["Kandra Smith"] = 56;
exits["a gloomy tunnel"] = dungeon;
167 words["!"] = 0;
```

- 在调用函数时，使用函数参数来动态分配属性名称：

```
var scores = {};
var updateScore = function (playerName, score) {
    scores[playerName] = score;
};
updateScore("Dax", 2000);
console.log(scores["Dax"]);
```

- 通过将一个对象作为参数传递给 Object.keys 方法来创建对象的键的数组：

```
var scores = {
    Kandra : 100,
    Dax : 60,
    Blinky : 30000
};
var keys = Object.keys(scores); // ["Kandra", "Dax", "Blinky"]
```

- 使用 split 方法将字符串拆分为数组，并将用于在字符串中确定断点位置的字符作为参数传递给它：

```
var query = "page=5&items=10&tag=pluto";
168 var options = query.split("&"); // ["page=5", "items=10", "tag=pluto"]
```

第二部分　组 织 代 码

对程序员而言，共享代码是司空见惯的事情。因为前辈们已经帮我们解决了许多问题，而我们对某些具体问题的解决方案也会在自己未来的项目中，或团队其他成员中，甚至更广泛的开发人员社区中得到应用。在共享的过程中，最重要的一点是明示其他人如何使用这些共享代码模块。具体方法是定义一个清晰的公共接口，或者定义一组能够供用户使用的属性和函数，而模块的内部构件（即程序如何运行）是非共享部分，必须得到保护，不能对他人开放。在本书第二部分，将讨论如何通过本地变量（而不是全局变量）的方法将某一模块实现隐藏起来。

随着程序越来越长，对代码进行组织优化的需求也随之增长。在一些比较长的程序中，我们发现有些模式和过程反复出现。本书第二部分将介绍如何在代码中通过模块的方法来提高程序的灵活性、重用性和可维护性。此外，还介绍了另一种提高程序灵活性的方法：使用 if 语句，只有在符合指定条件的情况下，才能执行某部分代码。

模块通常在一个项目中执行某些特定的、明确的任务。三种常见类型的模块分别是模型、视图和控制器。其中，模型用于表示和处理数据，视图用于在模型中呈现数据，控制器用于响应用户或系统动作并对模型和视图进行相应更新。

本书第二部分的最后一个内容是将游戏 The Crypt 拆分成模块，并更新游戏，使其包含诸多挑战，即玩家需要解决的难题。 169

第 11 章　作用域：隐藏信息

本章内容包括：

- 全局变量的危险性
- 局部变量的优势
- 使用命名空间来减少全局变量
- 使用函数创建局部变量
- 为用户返回接口

我们希望用户能够与基于控制台的程序进行交互，但又不想给用户太多控制权利！本章探讨了实现上述想法的具体方法：在控制台上既对用户隐藏部分程序；又对用户开放部分程序，他们可以定义被允许使用的属性和方法。例如，对于一个测验程序，用户应该能够提交答案，但是不能修改分数：

```
>quiz.submit("Ben Nevis")     // 允许用户这样做
>quiz.score = 1000000         // 不允许用户这样做
```

通过将公共接口与程序使用的私有变量和函数分离，程序开发者可以明确表明自己的态度，说明用户能够做什么，以及其他程序员在使用该代码时应该做什么，还可以降低代码被滥用的风险。这样做对程序，对玩家，以及对其他程序员都是有益的。 171

实现上述想法，函数是关键。在函数之外声明的变量可以在程序中的任何地方访问，而在函数内声明的变量只能在函数内访问。本章将介绍如何从那些允许用户访问其属性和方法的函数返回对象。构造函数也是函数，读者将看到如何使用特殊的 this 对象来设置公共属性和方法。

最后，我们将所学到的内容应用到 The Crypt 游戏，撤销用户对构造函数、位置和玩家属性的访问权限，提供显示玩家、位置信息以及导航地图的方法。允许用户进行以下动作：

```
>game.go("north") //移动位置
>game.get() //捡拾物品
```

禁止用户采用以下做法在游戏中作弊：

```
>player.health = 1000000
>player.place = exit
>player.items.push("The Mega-Sword of Ultimate Annihilation")
```

下面，我们首先看看在程序中随处可见的变量陷阱。

11.1 全局变量的危险性

在本书的第 2 章，我们就学习了如何声明、赋值和使用变量。下面的代码清单显示了一个非常简单的过程示例，在控制台上显示一座山峰的名字：

```
>Ben Nevis
```



代码清单 11-1 声明、赋值和使用变量
(http://jsbin.com/gujacum/edit?js,console)

```
var mountain;
mountain = "Ben Nevis";
console.log(mountain);
```

以上代码中，第一行声明变量 mountain，然后，可以在整个代码中使用该变量，我们为其分配了一个值，并将其作为参数传递给 console.log。

172 程序中的函数也可以访问 mountain 变量，如图 11-1 所示。

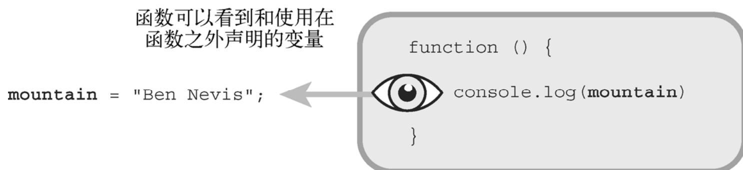


图 11-1 函数可以看到函数体外的 mountain 变量

下面的代码清单显示了使用 mountain 的 showMountain 函数，其输出结果与代码清单 11-1 相同。



代码清单 11-2 在函数内访问变量
(http://jsbin.com/zojida/edit?js,console)

```
var mountain = "Ben Nevis";   //在任何函数之外声明 mountain 变量
var showMountain = function () {
    console.log(mountain);     //即使在函数内也可以使用这个变量
};
showMountain();
```

像 mountain 这样，变量声明不在任何函数体内，因此在程序的任何地方都可以访问该变量，这样的变量称为**全局变量**。全局变量貌似非常好用：只需声明一次，就可在程序的任何地方自由使用它们。但不幸的是，它们有一些不足之处，而且通常认为是一些很严重的缺陷。下面，我们来研究避免使用这些全局变量的原因。

11.1.1 访问所有区域——偷窥和篡改

我们希望用户能够在控制台上与程序进行交互，但是又不想让他们窥探到程序中所有的变量。下面的代码清单显示了一个小测验。



代码清单 11-3　一个小测验
(http://jsbin.com/nubipi/edit?js,console)

```
var question = "What is the highest mountain in Wales?";
var answer = "Snowdon";
console.log(question);
```

因为 question 和 answer 是全局变量，所以在整个程序中和控制台上都可以访问和使用。一旦用户运行这个程序，在控制台提示符下，他们可以输入 answer，在按下〈Enter〉键后，直接将 answer 变量的值显示出来。用户就这样轻而易举地窥探到了我们极客的世界！ 173

```
>answer
Snowdon
```

这可不仅仅是简单的窥视，而是一个大问题，这意味着卑鄙的篡改！用户可以随意改变全局变量的值，他们就可以更新修改测验分数、银行余额、周转速率、商品价格和考试答案。

```
>answer = "Tryfan"
```

11.1.2　访问所有区域——借助于实现

有时候，不能立即实现的想法也很重要，我们需要一些时间才能够对其充分理解。如果有些想法并不是单击一次鼠标就能实现，也不必担心——读者可以在阅读本章的其他例子之后，随时返回来再次理解。

偷窥和修改不仅仅只是用户的问题，还有可能是程序员的问题（见下栏“谁在使用你的代码？”）。如果其他程序员使用你编写的代码，你可不希望他们在私下对你的代码进行修改，或者是以你的代码内部构件为基础来编写他们自己的代码。另外，当你发布一个内部使用了更高效算法的新版本代码时，依赖于这些变量的其他代码就会出问题。

谁在使用你的代码？

回忆一下，我们在第 7 章中创建了 spacer 函数，后来一直在 The Crypt 代码中使用这些函数。这就是一个我们使用自己创建代码的例子。

程序员通常属于某一个团队，团队各成员针对同一个程序的不同方面协同工作。没必要让每个成员分别创建自己的格式化函数，他们应该全都可以使用我们的 spacer 命名空间。

他们分享使用 spacer 命名空间后，对其赞美有加，还鼓励我们更广泛地分享 spacer 命名空间。于是我们将 spacer 命名空间代码上传到了社区代码库中，例如 Github.com（https://github.com）或 npmjs.com（https://www.npmjs.com），这样其他程序员就可以下载和使用该代码，甚至对它进行改进和扩展。

有一个伯乐看中了 spacer 代码，并对其进行了深入开发。现在，spacer 代码甚至有了自己的网站！该网站人气很旺，大家都在使用 spacer.js 库。

用户应该能够访问**接口**（即公开的功能），而不应该关心程序是如何**实现**的（即这些功能是如何编写代码的）。

接口与实现

接口是开发者希望用户访问的一组属性和函数，是这些属性和函数与应用程序的交
174 互，例如 ATM 上的屏幕、按钮和出钞口。

实现是使应用程序在幕后完成工作的代码。用户通常看不到它。例如，使 ATM 正常工作的代码包括：ATM 如何与银行通信以及 ATM 如何识别和清点钞票。

例如，要在 The Crypt 中实现地点之间的连接，可以对出口方向使用 exits 数组，对该出口所通往的目的地使用 destinations 数组。如下所示：

```
var exits = [];
var destinations = [];
```

然后，编写一个 addExit 函数来管理和添加新的出口。

```
var addExit = function (direction, destination) {
    exits.push(direction);
    destination.push(destination);
};
```

我们期望其他使用 Place 对象进行编程的程序员也能够使用 addExit 函数（图 11-2）。

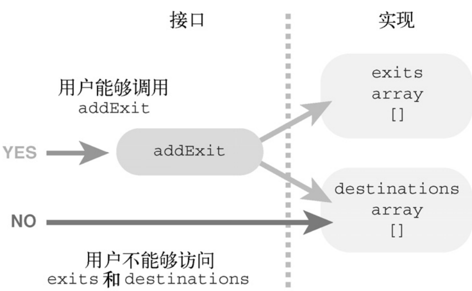


图 11-2　用户应该能够使用接口，而不能够使用实现

上图显示，addExit 是接口的一部分，是用户可以使用的属性和函数的集合。

```
addExit("north", kitchen);    //很好——用户在使用函数
```

但是如果用户可以访问 exits 和 destinations 变量，他们可能会绕过函数而直接使用这些变量。

```
exits.push("north");     //糟糕——用户在访问变量
175 destinations.push(kitchen);
```

目前，我们的工作进展顺利，其他程序员的编程工作也进展顺利。在重新查看代码后，我们决定通过使用单个对象来改进两个独立的数组。

```
var exits = {};
// destination 数组消失，在新的实现中不再需要 destination 数组
```

为了使用实现 exits 的新方法，我们对 addExit 函数进行了更新：

```
var addExit = function (direction, destination) {
   exits[direction] = destination;
};
```

使用该接口的程序员看不到他们的程序有任何变化。但是绕过接口直接访问变量的程序员发现他们的程序运行中断了！（见图 11-3）

```
addExit("north", kitchen);    //很好，照常运行
exits.push("north");          //错误！exits 不是数组
destinations.push(kitchen);   //错误！destinations 不存在
```

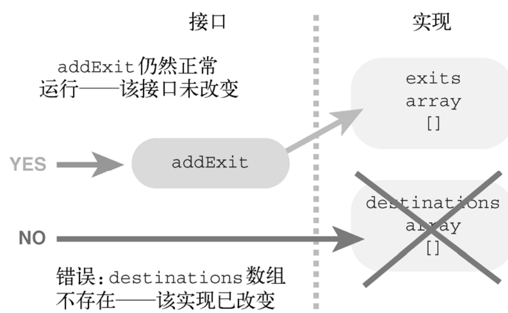


图 11-3　如果某实现更改了，依赖于该实现的代码会报错

以上可见，通过全局变量使我们能够访问程序所有区域的做法模糊了实现和接口之间的界限，赋予了用户偷窥和篡改的自由，并在实现的细节改变时导致了程序的崩溃。

11.1.3　命名冲突

因为程序可能是由不同的团队或程序员所编写的代码组成，所以很可能在多个地方使用了相同的变量名。例如，一个程序员声明一个变量名称 spacer（一个格式化函数的命名空间），随后该程序中的另一个程序员也声明了一个相同的变量名称 spacer（一个基于控制台的空间冒险游戏中的犬科角色），第二个变量声明就会覆盖第一个变量声明——发生冲突碰撞！为了避免以上碰撞，我们非常需要有一种方法能够保护变量免于彼此覆盖（第 13 章将更详细地探讨冲突碰撞问题）。 176

11.1.4　难以查找的错误

代码长度有可能成千上万行。依赖于全局变量的程序越庞大越弱不禁风，因为对全局变量的声明可能是在程序中很远的地方，并且在整个程序中这些全局变量都有可能被函数和指令所窥视和篡改。一旦程序出现问题（很可能会出现问题），我们将难以跟踪代码的流程，难以锁定问题的具体位置。

11.2　局部变量的优势

在函数体内声明的变量无法在该函数之外访问，这些在函数体内声明的变量称为**局部变**

量。局部变量只在声明该变量的函数体内局部发挥作用（图 11-4）。



图 11-4　在函数体内声明的变量是该函数的局部变量

如果在函数之外访问该函数的局部变量就会报错。代码清单 11-4 欲将 mountain 变量的值显示到控制台上，但是只会得到以下报错消息：

```
>ReferenceError: Can't find variable: mountain
```

不同浏览器可能会显示不同格式的报错信息。



代码清单 11-4　在控制台上无法显示的变量
(http://jsbin.com/bobilu/edit?js,console)

```
var hide = function () {
    var mountain = "Devils Tower";
};                          //在函数内声明 mountain 变量
console.log(mountain);   //尝试在该函数之外访问 mountain 变量
```

在所有函数之外声明的变量集合称为全局范围。每个函数创建自己的局部范围，即局部
177 变量集合。在代码清单 11-4 中，变量 mountain 不在函数之外的全局范围中，因此在 console.log 语句中使用该变量时会报告错误。

在同一范围内使用变量就不会产生任何麻烦，见以下代码清单 11-5，在控制台上可见：

```
>Devils Tower
```



代码清单 11-5　该变量在函数内部可见
(http://jsbin.com/raluqu/edit?js,console)

```
var show = function () {
    var mountain = "Devils Tower";
    console.log(mountain);     //在函数体内，将 mountain 变量作为一个参数传递给
                               //console.logas
};
show();
```

代码清单 11-6 将全局变量和局部变量组合在一起。程序可以在任何地方访问全局变量，但只能在 show 函数内部使用局部变量 secretMountain，在控制台上输出如下信息：

```
>Ben Nevis
>Devils Tower
>Ben Nevis
```

```
>ReferenceError: Can't find variable: secretMountain
```



代码清单 11-6　全局变量和局部变量
(http://jsbin.com/riconi/edit?js,console)

```
var mountain; //声明全局变量：mountain 和 show，在整个程序中可见
var show;
mountain = "Ben Nevis";
show = function () {
    var secretMountain= "Devils Tower";  //声明一个局部变量，secretMountain，
                                         //只在 show 函数中可见
    console.log(mountain); //使用 mountain 变量。没问题，因为它是全局变量。
    console.log(secretMountain);     //使用 secretMountain 变量。没问题，因为
                                     //在同一个局部范围内
};
show(); //调用 show 功能。没问题，因为 show 也是一个全局变量。
console.log(mountain);          //使用 mountain 变量。没问题，因为它是全局变量。
console.log(secretMountain);    //使用 secretMountain 变量。错误！因为它不在全
                                //局范围内
```
178

11.3　接口——控制访问权限并提供可用功能

我们希望用户能够在控制台上与程序进行交互，但是不希望用户挖掘程序的实现并对程序做出随意更改。向用户开放的一组属性和操作称为接口。下面我们就学习如何提供接口，同时隐藏其他内容。

在本节中，我们会使用一个非常简单的计数器程序。在第 11.4 节中，我们将开发一个测验应用程序的界面。在 11.5～11.7 节，我们将所学内容应用于游戏 The Crypt。

以下代码清单是计数器程序的第一个版本，使用全局变量保存当前计数。



代码清单 11-7　计数器
(http://jsbin.com/yagese/edit?js,console)

```
var counter = 0;   //声明一个全局变量 counter，并赋值
var count = function () {
    counter = counter + 1;
    return counter; //在函数内访问 counter
};
```

运行该程序，并在控制台提示符下按照下列步骤操作（用户操作的内容前面会显示提示符>，但程序响应的内容前面不会显示提示符>）。

```
>count()
1
>count()
2
>count()
```

```
3
```

程序看似运行正常，那还有什么问题？问题在于 counter 是一个全局变量，因此用户可以随时更改它，如下所示：

```
>counter = 99
99
>count()
100
```

我们并不想让用户拥有像这样的权力。下面，将使用 11.2 节中所介绍的局部变量知识
179 对用户隐藏 counter 变量。

11.3.1　使用一个函数来隐藏变量

针对计数器应用程序，我们希望实现以下两个目标：

- 对用户隐藏 counter 变量
- 对用户开放 count 函数

以下代码清单 11-8 显示了一个解决方案，运行该程序，可以在控制台上看到以下交互：

```
>count()
1
>counter
Can't find variable: counter
```

（再次提示，您收到的错误可能会略有不同，不同浏览器可能会显示不同格式的报错信息。）



代码清单 11-8　隐藏 counter 变量
(http://jsbin.com/fuwuvi/edit?js,console)

```
var getCounter = function () {
    var counter = 0;     //在 getCounter 函数内部声明一个局部变量 counter
    var countUpBy1 = function (){   //在 getCounter 里面，嵌套定义计数函数
                                    //countUpBy1
        counter = counter + 1;
        return counter;             //返回该计数函数，以备赋值
    };
    return countUpBy1;
};
var count = getCounter();           //将 getCounter 返回的计数函数赋值给
                                    //count 变量，以备后用
```

表 11-1 列出了一系列问题和解决方案，图 11-5 显示返回后赋值给 count 的计数函数仍然可以访问局部变量 counter。

表 11-1　计数器应用程序的问题和解决方案

问　　题	解决方案
对用户隐藏 counter 变量	在 getCounter 函数内声明 counter 变量。作为局部变量，用户不能访问 counter
函数 countUpBy1 的功能是使计数器递增，它需要访问 counter 变量	在 getCounter 函数内定义 countUpBy1 函数。因为 countUpBy1 在 getCounter 里面，countUpBy1 可以访问 counter 变量
希望用户能够调用计数函数	从 getCounter 函数返回计数函数 countUpBy1，然后可以将其分配给一个全局变量

180

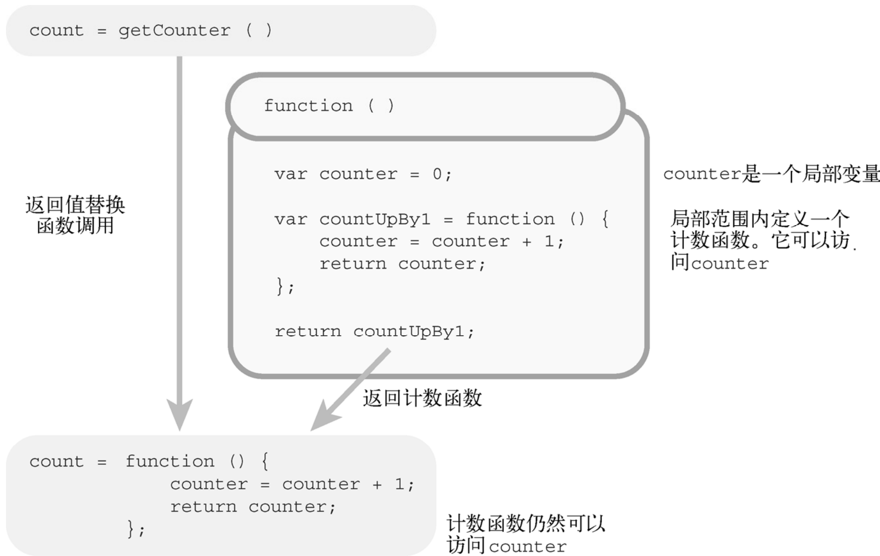


图 11-5　返回的函数仍然可以访问局部变量 counter

通过返回一个函数并将其分配给 count 变量（图 11-5），我们为用户提供了一种使用该程序的方法；我们给用户提供了一个接口——调用 getCounter()来获取一个计数函数，然后调用 count()来增加计数器。

11.3.2　使用 getCount 创建多个独立计数器

在以上代码清单 11-8 中，定义了一个函数 getCounter 来创建计数器。每次调用 getCounter 时，都会执行以下三个步骤：

（1）声明一个 counter 变量并赋值为 0。

（2）定义一个函数进行计数，该函数使用步骤（1）中的 counter 变量。

（3）返回计数函数。

每次调用 getCounter 时都会声明一个 counter 变量并定义一个计数函数。如果多次调用 getCounter，众多 counter 变量如何能做到不相互干扰？每次运行 getCounter 时，都会创建一个新的局部范围，我们得到的是 counter 变量的多个副本，每个变量都隔离在自己的变量集合中，也就是其局部范围内。

以下代码清单 11-9 对 getCounter 代码进行了更新，创建了两个计数器，在控制台上交互如下：

```
>climbCount()
1
>climbCount()
181 2
>climbCount()
3
>peakCount()
1
>climbCount()
4
```

可以看到，peakCount 和 climbCount 不会互相干扰。



代码清单 11-9　多个计数器
(http://jsbin.com/sicoce/edit?js,console)

```
var getCounter = function () { //定义计数函数并同时返回计数结果，不需要变量
    var counter = 0;
    return function () {
        counter = counter + 1;
        return counter;
    };
};
var peakCount = getCounter(); //多次调用 getCounter 以创建多个独立的计数器
var climbCount = getCounter();
```

可见，将 counter 定义为局部变量就可以拥有多个独立的计数器。

11.3.3　用构造函数创建多个独立的计数器

如果打算创建多个计数器，可以使用构造函数。如第 9 章所述，构造函数简化了使用相似属性和方法来创建对象的过程。使用 new 关键字可以调用构造函数，对于多个计数器，调用构造函数如下所示：

```
var peaks = new Counter();
var climbs = new Counter();
```

构造函数会自动创建一个对象，并将该对象分配给特殊的 this 变量，构造函数还会返回该对象。我们可以将计数函数设置为 this 的一个属性，使其在构造函数之外可用，详见以下代码清单 11-10。在控制台上，可以如下调用计数函数：

```
>peaks.count()
1
>peaks.count();
2
>climbs.count();
182 1
```



代码清单 11-10　计数器构造函数
(http://jsbin.com/yidomap/edit?js,console)

```
var Counter = function () {
   var counter = 0;
   this.count = function () {      //将计数函数指定为特殊的 this 对象的属性
      counter = counter + 1;
      return counter;
   };
};
var peaks = new Counter();
var climbs = new Counter();
```

对比代码清单 11-9 和 11-10，会发现二者几乎相同。唯一的区别在于，在代码清单 11-10 中，将计数函数设置为 this 的属性，而不是返回该函数。构造函数自动返回 this 对象，所以不需要手动去做。构造函数依然是一个函数，所以 counter 依然是一个局部变量。以上两种方法（使用普通函数和使用构造函数）都可以有效返回接口并隐藏实现的细节。

以上计数器是一个既简单又说明问题的好例子。下面我们来看一个包含更多互动部分的程序——测验应用程序。请注意，绝对不允许用户作弊。

11.4　创建一个简单的测验应用程序

这款简单的测验应用程序应该能够做到以下三点：

（1）在控制台上显示问题。

（2）在控制台上显示当前问题的答案。

（3）移到问题库中的下一个问题。

在控制台上进行交互的模样大致如下所示：

```
>quiz.quizMe()
What is the highest mountain in the world?
>quiz.showMe()
Qomolangma
>quiz.next()
Ok
>quiz.quizMe()
What is the highest mountain in Scotland?
```

可见，通过使用三个函数 QuizMe、showMe 和 next 可以满足该应用程序的三个要求。另外，我们还需要问题和答案数组以及一个变量来跟踪哪一个是当前问题。全局变量（在任何函数之外声明的变量）处于全局命名空间中。为了避免测验变量污染全局命名空间，可以将它们设置为单个对象的各个属性。正如在第 7 章中所见，可以使用单个对象把相关多个变量收集到一起，通常称为一个命名空间。 183

11.4.1　将对象用作一个命名空间

这款应用程序只创建唯一的全局变量 quiz，所有其他变量都设置为 quiz 对象的各个属性，如代码清单 11-11 所示。问题和答案数组如下所示：

```
var quiz = {
    questions: [
        {
            question: "What is the highest mountain in the world?",
            answer: "Qomolangma"
        }
    ]
};
```

然后，可以通过 quiz.questions 访问该数组。请注意，数组索引值从 0 开始，所以要访问第一个问题和答案对象，请使用 quiz.questions [0]。要获得 questions 数组的第一个问题和第一个答案，请分别使用以下形式：

```
quiz.questions[0].question;
quiz.questions[0].answer;
```

使用 qIndex 属性来跟踪当前问题。



代码清单 11-11　改进这款测验应用程序 (http://jsbin.com/tupoto/edit?js,console)

```
var quiz = {          //创建一个对象并将其分配给 quiz 变量
    questions: [      //创建一个数组并将其分配给 quiz 对象的 questions 属性
        {             //对每个问题和答案使用一个对象
            question: " What is the highest mountain in the world?",
                      //为数组中的每个对象设置一个 question 属性和一个
                      //answer 属性
            answer: "Qomolangma"
        },
        {
            question: "What is the highest mountain in Scotland?",
            answer: "Ben Nevis"
        },
        {
            question: "How many munros are in Scotland?",
            answer: "284"
        }
    ],
    qIndex: 0,                       //创建一个 qIndex 属性来跟踪当前的问题
    quizMe: function () {            //使用 quizMe 显示当前问题
        return quiz.questions[quiz.qIndex].question;
    },
    showMe: function () {            //使用 showMe 显示当前答案
```

184

```
        return quiz.questions[quiz.qIndex].answer;
    },
    next: function () {              //使用 next 移动到下一个问题
        quiz.qIndex = quiz.qIndex + 1;
        return "Ok";
    }
};
```

以上代码清单 11-11 满足了上述应用程序的三个要求，并且只使用了一个全局变量 quiz，程序所需要的全部变量都包括在 quiz 对象内，作为 quiz 的属性来访问，因此它们全都属于 quiz 命名空间。

很遗憾，这款应用程序还不能克服全局变量的一个缺陷：所有区域均可访问（参见第 11.1.1 节）。quiz 对象的所有属性都是公开的——用户仍然可以窃取最高分并修改所有数值。

11.4.2　隐藏问题数组

在以上代码清单 11-11 中，用户可以在控制台上访问 quiz 对象的所有属性。这样，他们就有可能恶作剧，改变各属性，如下所示：

```
>quiz.qIndex = 300
>quiz.questions[2].answer = "282"
```

其实，我们希望用户只能使用 quiz.quizMe、quiz.showMe 和 quiz.next。下面通过使用局部变量来实现上述目的，使 questions 数组和 qIndex 的值变为私有，如图 11-6 所示。

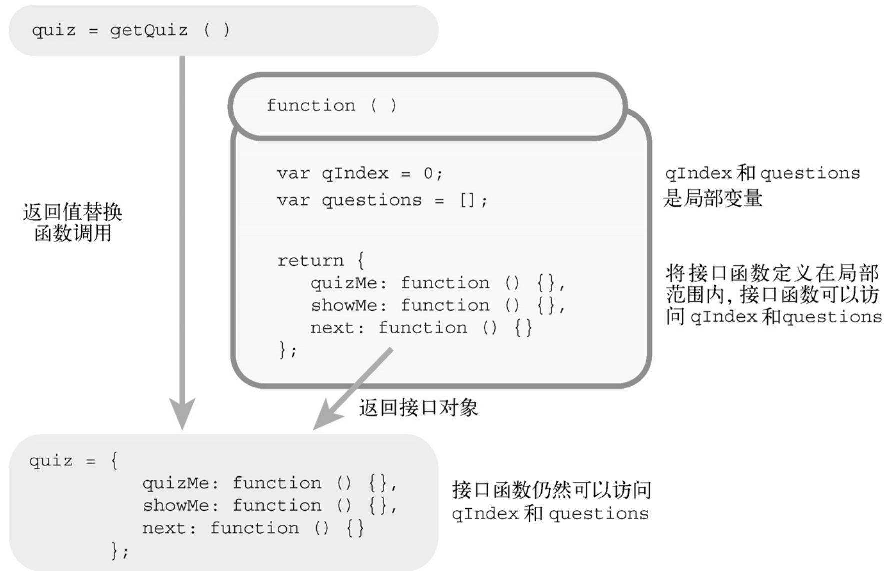


图 11-6　使用局部变量使 qIndex 和 questions 变为私有　185

以下代码清单 11-12 中使用 getQuiz 函数创建一个局部范围，用于隐藏 questions 数组和 qIndex 的值。



代码清单 11-12　隐藏问题和答案
(http://jsbin.com/qahedu/edit?js,console)

```
var getQuiz =function () {    //使用一个函数创建局部作用域
    var qIndex = 0;           //在函数内部声明局部变量
    var questions = [
        {
            question: " What is the highest mountain in the world?",
            answer: "Qomolangma"
        },
        {
            question: "What is the highest mountain in Scotland?",
            answer: "Ben Nevis"
        },
        {
            question: "How many munros are in Scotland?",
            answer: "284"
        }
    ];
    return {         //返回一个对象，该对象具有用户可以访问的属性
        quizMe :function () {
        return questions[qIndex].question;
        },
        showMe :function () {
            return questions[qIndex].answer;
        },
        next :function () {
            qIndex = qIndex + 1;
            return "Ok";
        }
    };
};
var quiz = getQuiz();    //调用函数，将返回的对象分配给 quiz 变量
```

该程序返回一个对象，该对象具有三个属性：quizMe、showMe 和 next。在程序的最后一行，将返回的对象分配给 quiz 变量，用户可以使用 quiz 访问这三个函数：

```
>quiz.quizMe()
What is the highest mountain in the world?
>quiz.answer()
Qomolangma
```

我们将能够使上述测验应用程序工作的代码称为该程序的实现，通过使用局部变量可以
186 将某些实现隐藏在 getQuiz 函数中。该函数返回的对象为用户提供了一个接口，接口是一种用户与程序公开交互的方式。用户可以调用 quizMe、showMe 和 next 函数，因为它们都是接口对象的方法。用户无法访问 qIndex 和 questions 变量，因为它们都是 getQuiz 函数内的局部

变量。

图 11-7 显示了用户在 JS Bin 控制台提示符下尝试访问变量 qIndex 和 questions 时，系统报错的界面（因浏览器不同，错误提示信息可能略有不同）。

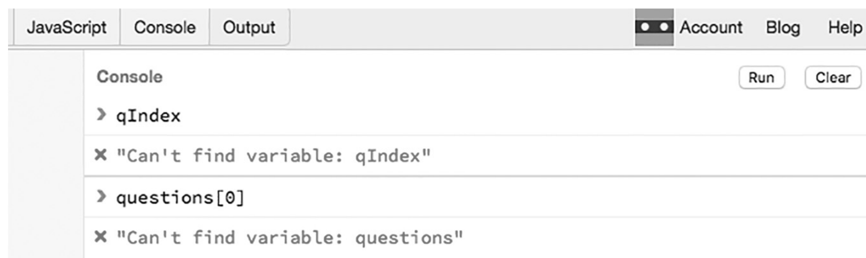


图 11-7　在控制台上访问 qIndex 和 questions 会导致错误

11.5　The Crypt——隐藏玩家信息

在 11.5～11.7 节中，我们将更新 The Crypt 代码，隐藏一部分变量、属性和函数，这些隐藏的内容属于该程序的实现，是程序开发者不愿意对外界公开的部分，同时，也会考虑该程序的接口应该采取什么形式向玩家和其他程序员公开。

11.5.1　当前的 Player 构造函数——全部内容都公开

到目前为止，我们还没有控制用户对 Player 对象中数据的访问。以下代码清单显示了 Player 构造函数的当前形式。



代码清单 11-13　Player 构造函数 (http://jsbin.com/dacedu/edit?js,console)

```
var spacer = { /*  在 JS Bin 上可见  */ };
var Player = function (name, health) {
   var newLine = spacer.newLine();
   this.name = name;    //在 this 对象上设置属性使所有数据可用
   this.health = health;
   this.items = [];
   this.place = null;
   this.addItem = function (item) { … };//在 this 对象上设置属性使所有方法可用
   this.getNameInfo = function () { … };
   this.getHealthInfo = function () { … };
   this.getPlaceInfo = function () { … };
   this.getItemsInfo = function () { … };
   this.getInfo = function (character) { … };
   this.showInfo = function (character) { … };
};
```

187

我们将所有玩家数据和所有函数都设置为特殊的 this 对象的属性。当使用 new 关键字调

用 Player 构造函数时，Player 构造函数会自动创建 this 对象。以下语句就是调用 Player 构造函数来创建一个新玩家：

```
var player1 = newPlayer("Jahver", 80);
```

构造函数自动返回 this 对象，并将该对象分配给 player1 变量。因为在构造函数中已经包含了所有内容，返回并分配给 player1，现在就可以使用 player1 来访问该对象的数据和方法，例如 player1. name、player1. items、player1. addItem、player1. getInfo 和 player1. showInfo。

11.5.2 更新版的 Player 构造函数——某些变量被隐藏

为了控制用户对玩家数据的访问，可以使用参数和变量，而放弃上述为 this 对象属性赋值的做法。函数定义中圆括号内的参数与函数体内声明的变量作用域相同，都是该函数内的局部变量，可以在该函数体内的任何地方使用，但不能在该函数之外访问。

```
var Player = function (name, health) {
    // name 和 health 都是局部变量
    // name 和 health 在这里可以使用
    this.getHealthInfo = function () {
        // name 和 health 在这里可以使用
    };
};
// name 和 health 在这里不能使用
//这里是 player 函数体之外的区域
```

图 11-8 显示了两个版本的 Player 构造函数，旧版对应代码清单 11-13，新版对应代码清单 11-14，二者的变化在于一些变量和属性在函数体外是否可见。新版 Player 构造函数隐藏
188 了局部作用域中的大多数函数，仅将 4 个函数作为接口分配给特殊的 this 对象。

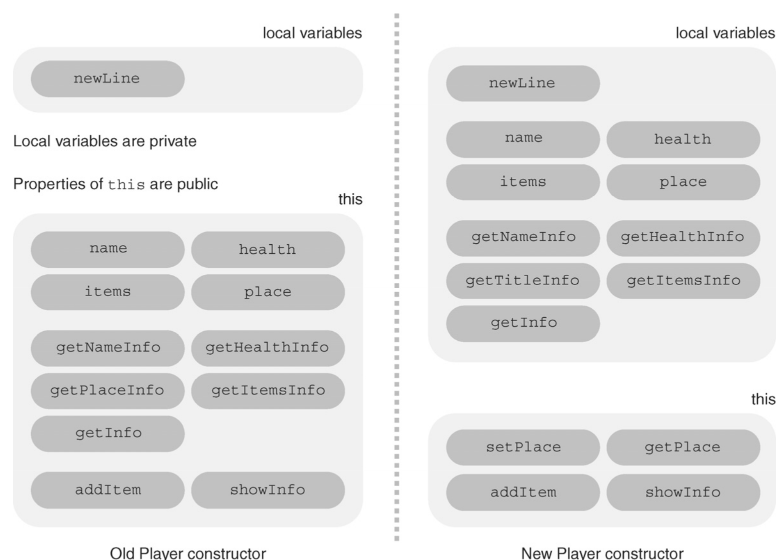


图 11-8　将一些变量和函数隐藏在构造函数的局部作用域内

以下代码清单 11-8 显示了在新版代码中的更新部分，使用 var 声明的局部变量阻止玩家对数据的直接访问。



代码清单 11-14　在构造函数中隐藏玩家信息 (http://jsbin.com/fuyaca/edit?js,console)

```
var spacer = { /* 在JS Bin上可见 */ };
var Player = function (name, health) {    //在this对象上设置属性使所有方法可用
    var newLine = spacer.newLine();
    var items = [];
    var place = null;
    var getNameInfo = function () { //将这些函数私有化——以供内部使用
        return name;
    };
    var getHealthInfo = function () {
        return "(" + health + ")";
    };
    var getItemsInfo = function () {
        var itemsString = "Items:" + newLine;
        items.forEach(function (item, i) {
            itemsString += "   - " + item + newLine;
        });
        return itemsString;
    };
    var getTitleInfo = function () {    //将这些函数私有化——以供内部使用
        return getNameInfo() + " " + getHealthInfo();
    };
    var getInfo = function () {
        var info = spacer.box(getTitleInfo(), 40, "*");
        info += "  " + getItemsInfo();
        info += spacer.line(40, "*");
        info += newLine;
        return info;
    };
    this.addItem = function (item) {
        items.push(item);
    };
    this.setPlace = function (destination) { //添加方法来管理对place变量的访问
        place = destination;
    };
    this.getPlace = function () {
        return place;
    };
    this.showInfo = function (character) {
        console.log(getInfo(character));
    };
};
```

189

从以上代码可见，我们把不希望在构造函数之外看到的属性和函数分配给局部变量，把希望在构造函数之外看到的方法分配给 this 对象的属性。

```
var items = [];                        // 私有化
this.showInfo = function () { };    // 公开化
```

以上代码中，setPlace 和 getPlace 方法可以让用户访问 place 变量。既然这两个方法允许用户访问 place 变量，为何要使用一个变量来使其私有？因为采用这种方式可以为用户提供一个接口，同时可以让开发者将程序实现隐藏起来，并可以使开发者对用户的访问有所调控：当调用 setPlace 方法，将 destination 参数分配给 place 变量之前，可以先检查 destination 参数是否为有效位置；当调用 getPlace 方法，在返回该地点之前，可以检查该用户是否有权访问。目前我们尚未将这两项额外的检查落实到位，但是当日后需要添加任何检查条件时，这两种方法就会立刻派上用场。

以上代码中，通过使用 showInfo 简化了对信息的显示，不再显示当前位置。Place 对象
190 具有自己的 showInfo 方法，因此无需将已经包含在玩家信息内的地点详情再次重复显示。
现在，当用户在控制台上查看玩家信息时，会发现内容减少了，如下所示：

```
****************************************
*  Kandra (50)                         *
****************************************
Items:
-The Sword of Doom
****************************************
```

在玩家名字后面的括号中显示了玩家的健康值。如果需要显示玩家当前的位置信息，可以调用 getPlace 来检索 place 对象，然后调用 place 对象的 showInfo 方法：

```
var place = player.getPlace();
place.showInfo();
```

11.6 The Crypt——隐藏地点信息

本节中对 Place 构造函数所做的更新与 11.5 节中对 Player 构造函数所做的更新完全相同：隐藏某些数据并提供按需访问的方法。以下代码清单 11-15 与第 10 章中的 Place 构造函数的结构相同，更新之处是在 this 对象上设置了数据和方法。



代码清单 11-15 Place 构造函数
(http://jsbin.com/kavane/edit?js,console)

```
var spacer = { /*  在 JS Bin 上可见 */ };
var Place = function (title, description) {
    var newLine = spacer.newLine();
    this.title = title;  //使用属性来保存地点数据
    this.description = description;
    this.items = [];
```

```
    this.exits = {};
    this.getItemsInfo = function () { }; //创建函数来构建信息字符串
    this.getExitsInfo = function () { };
    this.getTitleInfo = function () { };
    this.getInfo = function () { };
    this.showInfo = function () { };     //创建函数来更新并显示地点数据
    this.addItem = function (item) { };
    this.addExit = function (direction, exit) { };
};
```

以下代码清单 11-16 是更新版的 Place 构造函数：使用参数和变量来隐藏数据，一个新的 getExit 函数返回指定方向目的地，还有一个新的 getLastItem 方法从 items 数组返回最后一项元素。 191



代码清单 11-16　在构造函数中隐藏地点信息
(http://jsbin.com/riviga/edit?js,console)

```
var spacer = { /*  在 JS Bin 上可见 */ };
var Place = function (title, description) {   //使用参数和 var 关键字创建私
                                              //有变量
    var newLine = spacer.newLine();
    var items = [];
    var exits = {};
    var getItemsInfo = function () {     //将这些函数私有化，仅供内部使用
    var itemsString = "Items: " + newLine;
    items.forEach(function (item) {
          itemsString += " - " + item;
          itemsString += newLine;
       });
    return itemsString;
    };
    var getExitsInfo = function () {
    var exitsString = "Exitsfrom " + title;
       exitsString += ":" + newLine;
       Object.keys(exits).forEach(function (key) {
          exitsString += " - " + key;
          exitsString += newLine;
       });
       return exitsString;
    };
    var getTitleInfo = function () {
       return spacer.box(title, title.length + 4, "=");
    };
    var getInfo = function () {
       var infoString = getTitleInfo();
       infoString += description;
```

```
        infoString += newLine + newLine;
        infoString += getItemsInfo() + newLine;
        infoString += getExitsInfo();
        infoString += spacer.line(40, "=") + newLine;
        return infoString;
    };
    this.showInfo = function () {
        console.log(getInfo());
    };
    this.addItem = function (item) {
        items.push(item);
    };
    this.addExit = function (direction, exit) {
        exits[direction] = exit;
192 };
    this.getExit = function (direction) {  //添加一个公共方法来访问目的地
        return exits[direction];
    };
    this.getLastItem = function () {      //添加一个公共方法从 items 数组中获
                                          //取最后一项元素
        return items.pop();
    };
};
```

以上代码中，使用 Player 和 Place 构造函数隐藏了某些属性和函数，同时制作了一小套方法(即接口)，这些方法通过 this 对象可见。该程序中，以上两个构造函数之外的其他部分只能使用接口中的方法，如果其他程序员在他们的冒险游戏程序中使用本程序，也只能使用接口中的方法。

那些在控制台上输入命令的玩家，无意中发现自己竟然能够修改一些重要的值，难免心中窃喜。现在我们完全可以阻止他们的这种过失行为（说作弊就太难听了），从而也能保持游戏的乐趣。我们只给玩家提供一小套简单的行动，例如探险、收集、猛击和摧毁等一系列适合本游戏的动作。

11.7 The Crypt——用户交互

在第 10 章中，我们已经创建了游戏 The Crypt 的工作版本，可以让玩家从一个地方移动到另一个地方，然后拾起他们发现的物品，遗憾之处在于程序中使用的全局变量污染了全局命名空间。当然，我们也不必过于懊恼，那时只是年少无知。在本章，我们学会了一种新的方法来解决问题：对用户隐藏程序的实现。

以下代码清单 11-17 显示了程序实现的概要，完整版本在 JS Bin 上可见。它与第 10 章的版本几乎相同，不同之处在于使用了 getPlace 和 setPlace 玩家方法以及本章更新版构造函数中的 getExit 和 getLastItem 地点方法。更新部分的代码如下所示。



代码清单 11-17　游戏包含很多全局变量
(http://jsbin.com/dateqe/edit?js,console)

```
// spacer 命名空间
var spacer = { /* 格式化函数 */ };
// 构造函数
var Player = function (name, health) { … };
var Place = function (title, description) { … };
// 游戏控制函数
var render = function () {
    console.clear();
    player.getPlace().showInfo();
    player.showInfo();
};
var go = function (direction) {
    var place = player.getPlace();
    var destination = place.getExit(direction);
    player.setPlace(destination);
    render();
    return "";
};
var get = function () {
    var place = player.getPlace();
    var item = place.getLastItem();
    player.addItem(item);
    render();
    return "";
};
// 地图
var kitchen = newPlace("The Kitchen", "You are in a kitchen…");
var library = newPlace("The Old Library", "You are in a library…");
kitchen.addItem("a piece of cheese");
library.addItem("a rusty key");
kitchen.addExit("south", library);
library.addExit("north", kitchen);
// 游戏初始化
var player = newPlayer("Kandra", 50);
player.addItem("The Sword of Doom");
player.setPlace(kitchen);
render();
```

193

请注意，我们在程序中使用的全局变量：spacer 命名空间、Player 构造函数、Place 构造函数、游戏控制函数、所有的地点和玩家，以上这些变量都是在函数体之外进行赋值的。这样，全局污染就实在太严重了，简直是臭气熏天！在上述所有变量中，其实我们只需将游戏控制函数 go 和 get 提供给玩家，其他部分都可以隐藏。

11.7.1 接口——go 和 get

目前，用户只需要以下两项操作：

```
game.go("north");  //向当前位置的北方移动
game.get();   //从当前位置拾取一个物品
```

首先，我们需要隐藏程序的实现，可以将该游戏代码包裹在一个函数内。其次，为了授权用户进行上述两项操作，使用两个方法从函数返回一个接口对象：

```
return {
    go: function (direction) {
        // 移动到一个新地点
    },
    get: function () {
        // 拾取一个物品
    }
194 };
```

以上接口方法通过使用 player 和 place 对象功能来实现，这两个对象由它们各自的构造函数所创建。

11.7.2 隐藏程序的实现

为了隐藏程序的实现，程序只返回接口对象，把其余需要对用户隐藏的代码都包裹在函数中，以创建局部作用域。以下代码清单 11-18 突出显示了更新部分，完整代码在 JS Bin 上可见。



代码清单 11-18　让用户通过一个接口对象与游戏交互 (http://jsbin.com/yuporu/edit?js,console)

```
var getGame =function() {      //将实现包裹在一个函数中以创建局部作用域
    var spacer = { … };         //包括 spacer、Player 构造函数和 Place 构造函数
    var Player = function (name, health) { … };
    var Place = function (title, description) { … };
    var render = function () { … };//包括更新控制台的函数
    var kitchen = newPlace("The Kitchen", "You are in a kitchen…");
    var library = newPlace("The Old Library", "You are in a library…");
    kitchen.addItem("a piece of cheese");
    library.addItem("a rusty key");
    kitchen.addExit("south", library);
    library.addExit("north", kitchen);
    var player = newPlayer("Kandra", 50);      //创建一个玩家（它是局部的），
                                                //并给他一个始发地点
    player.addItem("The Sword of Doom");
    player.setPlace(kitchen);
    render(); //在控制台上显示初始信息
    return {  //返回一个对象作为接口，玩家可以访问其方法
```

```
            go: function (direction) {
                var place = player.getPlace();
                var destination = place.getExit(direction);
                player.setPlace(destination);
                render();
                return "";
            },
            get: function () {
                var place = player.getPlace();
                var item = place.getLastItem();
                player.addItem(item);
                render();
                return "";
            }
        };
    };
    var game = getGame();     //调用 getGame 函数，返回接口对象，并将其分配给 game 变量
```
195

以上代码的最后一条语句调用 getGame 函数，运行该函数然后返回一个接口对象，该对象具有 go 和 get 方法，这条语句将该接口对象分配给 game 变量。玩家可以通过 game 变量加圆点运算符来访问 go 和 get 方法，如下所示：

```
game.get();
game.go("east");
```

从以上代码可见，玩家无法访问游戏中的任何其他变量或函数。我们的目标实现了，至少在本章算是大功告成了！

如上例所示，在返回接口对象的函数中打包代码的过程（简称打包-返回过程），我们称之为模块模式。在编程实践中，我们常常将接口与实现分离开来，代码打包有助于提高代码的可移植性和重用性。在后续章节中，读者还会看到很多模块模式的代码，尤其在第 13 章，会看到很多使用 HTML 中的 script 标签导入模块的示例。在学习第 13 章内容之前，请在第 12 章学习如何做决定。If：如果你准备好了，请继续阅读；else：否则，请先休息放松一下，做做 Further Adventures 中的小练习，然后穿好盔甲，整装待发。

11.8　本章小结

■ 减少程序中全局变量的数量。全局变量是指在所有函数之外声明的变量，我们可以在程序中的任何地方访问全局变量，但这样做会污染全局命名空间，会暴露程序的实现，会引起命名冲突，还会导致程序中的错误难以发现：

```
var myGlobal = "Lookatme";
var useGlobal = function () {
    console.log("I can see the global: " + myGlobal);
};
console.log("I too can see the global: " + myGlobal);
```

- 将相关变量和函数收集在一起作为某一单个对象的各个属性，可以减少全局变量的数量。这样的对象通常称为命名空间：

```
var singleGlobal = {
    method1 : function () { … },
    method2 : function () { … }
};
```

- 将代码包裹在函数中以创建局部作用域，即局部变量的集合：

```
var getGame = function () {
    var myLocal = "A local scope, for local people.";
    console.log(myLocal); // 显示信息
};
console.log(myLocal); // 错误！此处 myLocal 未声明
196 // 它不在全局作用域
```

- 将代码包裹在函数中可以使变量私有化，可以隐藏实现，可以避免命名冲突，还可以降低错误难以发现的风险。
- 从函数返回一个公共接口，可以清楚地授权用户可以访问的属性和方法（模块模式）。
- 返回单个函数：

```
var getCounter = function () {
    var counter = 0; // 私有
    return function () { … };
};
var count = getCounter();
```

- 返回一个对象：

```
var getGame = function () {
    // 游戏的实现
    return {
        // 接口的方法
    };
};
var game = getGame();
```

- 请记住，构造函数如同任何其他函数一样可以创建局部作用域，并自动返回一个特殊的 this 变量作为公共接口：

```
var Counter = function () {
    var counter = 0; // 私有
    this.count = function () { … };
197 };
```

第 12 章　条件：有选择地运行代码

本章内容包括：

- 使用比较运算符
- 检查条件是真（true）还是假（false）
- if 语句 —— 仅在满足条件时运行代码
- else 子句 —— 当条件不满足时运行代码
- 确保用户输入的数据不会破坏程序代码
- 使用 Math.random()生成随机数

到目前为止，我们编写的所有代码都遵循单一运行路径，当调用函数时，函数体中的每条语句都将按顺序执行。虽然我们已经成功地完成了很多工作，也掌握了 JavaScript 的很多核心概念，但是我们编写的程序还缺乏灵活性，不能有选择地执行某些代码块。

在本章中，读者将学习如何运行那些满足指定条件的程序代码。这样，程序就拥有了分支执行的能力，为程序提供了选择性、灵活性和丰富性。在水果忍者游戏中，如果玩家切割了一个金橘，就可以得分；在 The Crypt 游戏中，如果用户指定了一个有效的方向，就可以移动到新位置；在微博中，如果用户的帖子长度少于 141 个字符，就可以将该文发布到微博上。 198

如果想知道程序如何做出决定，请继续读下去，因为下面的内容真的很有用！

12.1　有条件地执行代码

首先，我们来创建一个简单程序，请用户猜出密码数字。如果猜测正确，程序就会输出“Well done！”，在控制台上的交互如下所示：

```
>guess(2)
undefined
>guess(8)
Well done!
undefined
```

在控制台上怎么出现了讨厌的 undefined？当用户在控制台上调用函数时，将执行该函数体的代码，并显示返回值。但是在代码清单 12-1 中，因为 guess 函数中没有包含一条可以返回值的语句，因此该函数就自动返回了 undefined。



代码清单 12-1　猜数字
(http://jsbin.com/feholi/edit?js,console)

```
var secret = 8;                          //将一个数字赋值给 secret 变量
var guess=function(userNumber){          //定义一个函数来接收用户输入的数字并将该数
                                         //字分配给 guess 变量
   if (userNumber ===secret) {           //判断用户输入的数字是否与密码数字匹配
      console.log("Well done!");//如果这两个数字匹配，则在控制台上输出“Well done!”
   }
};
```

在以上代码中，guess 函数检查用户输入的数字是否与密码数字相等。guess 函数使用了严格相等运算符===和 if 语句，只有在数字完全匹配时才显示“Well done!”消息。下面将详细描述严格相等运算符和 if 语句，并介绍 else 子句。

12.1.1　严格相等运算符

严格相等运算符（===）用于比较两个值。如果二者相等，则返回 true；如果二者不相等，则返回 false。读者可以在控制台上进行测试：

```
>2 === 8
false
>8 === 8
true
>8 === "8"
false
> "8" === "8"
199 True
```

在以上第三个例子中，可以看到严格相等运算符判断数字 8 和字符串“8”是不相等的，这是因为数字和字符串是两种类型的数据。而 true 和 false 是第三种数据类型，称为布尔值。事实上，布尔值只有 true 和 false，无其他。在决定程序下一步应该做什么时，我们常常使用布尔值，例如下一节的 if 语句。

12.1.2　if 语句

使用 if 语句意味着只有当指定条件满足时才能执行代码块，如下所示：

```
if (条件) {
   // 需要执行的代码
}
```

如果圆括号中的条件为真（true），那么 JavaScript 执行花括号之间的代码块语句。如果条件为假（false），那么 JavaScript 跳过该代码块。请注意，在 if 语句表示结尾的花括号后面没有分号。

代码清单 12-1 使用严格相等运算符返回一个 true 或 false 值。

```
if (userNumber === secret) {
    console.log("Well done!");
}
```

在以上代码中，只有当用户输入的值 userNumber 等于 secret 的值时，才会将“Well done!”消息输出到控制台上。例如，secret 的值是 8，用户输入的值是 2。

```
if (2 === 8) {                      //这个条件是 false
    console.log("Well done!");      //这条语句不会执行
}
```

如果用户输入 8, 那么 if 语句就变成：

```
if (8 === 8) {                      //这个条件是 true
    console.log("Well done!");      // 这条语句会执行
}
```

12.1.3　else 子句

在 12.1.2 节中，我们学习了 if 语句，如果条件为 false，那么 JavaScript 会跳过该代码块。但是有些时候，如果 if 语句中条件的值为 false，还需要执行不同的代码，则可以通过在 if 语句中附加一个 else 子句来实现（图 12-1）。 200

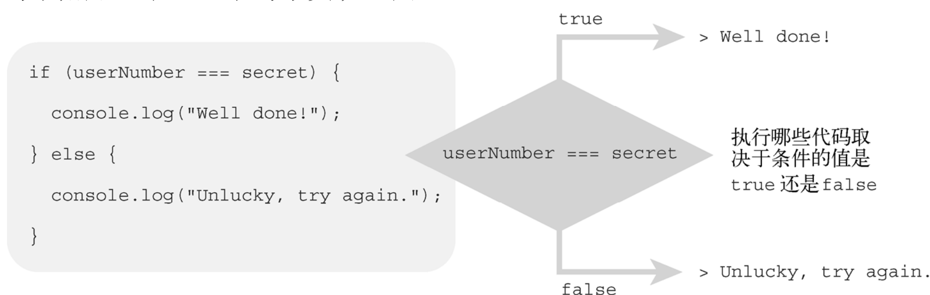


图 12-1　使用 if 和 else，根据条件的值来执行不同代码

以下代码来自代码清单 12-2：

```
if (userNumber === secret) {
    console.log("Welldone!");
} else {
    console.log("Unlucky, try again.");
}
```

如果 userNumber 和 secret 相等，JavaScript 将显示“Well done!”；否则，显示“Unlucky，try again”。请注意，在 else 子句表示结尾的花括号后面没有分号。例如，secret 的值为 8，用户输入的值是 2：

```
if (2 === 8) {                      //条件的值是 false
    console.log("Well done!");      //不执行该语句
```

```
} else {
    console.log("Unlucky, try again.");  //执行该语句
}
```

如果用户输入的值是 8：

```
if (8 === 8) {                            //条件的值是 true
    console.log("Well done!");            //执行该语句
} else {
    console.log("Unlucky, try again.");  //不执行该语句
}
```

这个猜密码数字的游戏在控制台上交互如下：

```
>guess(2)
Unlucky, try again.
undefined
>guess(8)
Well done!
201 undefined
```



代码清单 12-2　猜数字——else 子句 (http://jsbin.com/nakosi/edit?js,console)

```
var secret = 8;
var guess = function (userNumber) {
    if (userNumber === secret) {     //添加一个 if 语句，在圆括号中加入判断条件
        console.log("Well done!");   //仅当条件为 true 时执行此代码
    } else {                         //当条件是 false 时添加 else 子句
        console.log("Unlucky, try again.");    //仅当条件为 false 时才执行此代码
    }
};
```

下面，我们学习如何使用局部变量存放密码数字。

12.1.4　隐藏函数中的密码数字

在代码清单 12-2 中，secret 和 guess 变量声明在任何函数之外。我们在第 11 章中已经学过，这样做就意味着它们是全局变量，任何人无论在控制台上还是在整个程序中都可以访问它们。这样做对 guess 来说是非常好的——我们就是希望用户能够猜测数字；但是，对 secret 来说却是一个灾难——用户可以随意窥视和修改它的值。如果运行代码清单 12-2，用户就可以在控制台上执行下列操作：

```
>secret              //可以访问 secret，因为它是一个全局变量
8
>guess(8)
Well done!
Undefined
>secret = 20         //可以将 secret 重置为任何值
```

```
20
>guess(20)
Well done!
Undefined
```

用户居然可以随意设置 secret 的值！显然，这已经不再是一个猜数字的游戏了。

在本书第 11 章，讨论了如何在 JavaScript 中使用函数来创建局部作用域，即只有在函数内才能访问的变量集合。在代码清单 12-3 中，我们使用 getGuesser 函数来隐藏密码数字，并将 getGuesser 函数的返回值分配给 guess 变量（图 12-2）。

guess 是一个全局变量，可以供用户在控制台上访问：

```
>guess(2)
Unlucky, try again
Undefined
```
202

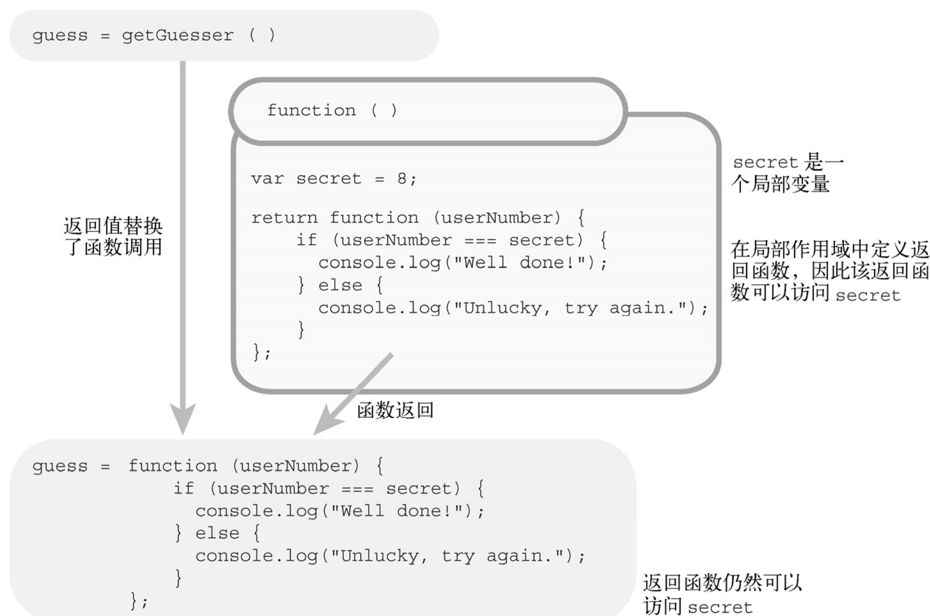


图 12-2　将 getGuesser 函数的返回值分配给 guess 变量



代码清单 12-3　猜数字——使用局部作用域 (http://jsbin.com/hotife/edit?js,console)

```
var getGuesser = function () {      //使用函数创建局部作用域
    var secret = 8;                 //将密码数字隐藏在局部作用域内
    return function (userNumber) {  //返回一个用户可以使用的函数来猜数字
        if (userNumber === secret) {     //判断用户的猜测是否与密码数字相同
            console.log("Well done!");  //只有数字匹配时才输出"Well done!"
        } else {
            console.log("Unlucky,try again."); //……否则输出"Unlucky,try again"
        }
    };
```

```
};
var guess = getGuesser();     //调用 getGuesser 并将返回函数分配给 guess 变量
```

在以上代码中，分配给 getGuesser 的函数创建了一个局部作用域，将 secret 变量置于其中，用户无权访问。该函数返回了另一个让用户猜密码数字的函数，并分配给 guess 变量。因为猜密码数字函数是在 getGuesser 函数创建的局部作用域中定义的，所以猜密码数字函数
203 可以访问 secret 变量并且能够进行密码猜测。

以上代码已经可以称得上是一款简单的猜谜游戏了，但是仅限于猜中那个一成不变的密码数字，实在是不太好玩！下面，我们使用 JavaScript 的 Math 命名空间中的几个方法，为猜谜游戏添加一些额外的神秘色彩！

12.2 利用 Math.random()生成随机数

Math 命名空间为我们提供了一种可以生成随机数的方法，该方法返回一个大于或等于 0 且小于 1 的数字。在控制台的提示符后输入 Math.random()，得到了以下随机数字：

```
>Math.random()
0.7265986735001206
>Math.random()
0.07281153951771557
>Math.random()
0.552000432042405
```

显然，这些密码数字是各不相同的，因为它们是随机的。除非你真的热衷于玩猜谜游戏，并且有很多空闲时间，否则要猜中这些随机数字可能是太难了。

我们需要将这些密码数字改良一下，把它们缩放到我们想要的范围内，然后将它们转换为整数。因为最初它们的初始值都小于 1，所以乘以 10 将使它们变成小于 10 的随机数。请看下面几条 Math.random 的赋值语句：

```
var number = Math.random();             // 0<= number<1
```

要缩放这些数字，请使用乘法运算：

```
var number = Math.random() * 10;        // 0<= number<10
```

要增大或缩小这些数字，请使用加法或减法：

```
var number = Math.random() + 1;         // 1<= number<2
```

先乘除后加减：

```
var number = Math.random() * 10 + 1;    // 1<= number<11
```

以上最后一次赋值得到的数字应该在 1 和 11 之间，可以等于 1，但小于 11。<=符号表示小于或等于，而<符号则表示小于。不等式 0<= number<1 表示数字在 0 和 1 之间，可以等于 0，但不能为 1（见第 12.3.1 节）。

通过上述语句，我们已经把随机数放大了，但是它们在控制台上的模样看上去并不理想：

```
>Math.random() * 10 + 1
3.2726867394521832
>Math.random() * 10 + 1
9.840337357949466
```

204

下一步，我们要去掉每个数字的小数部分，将以上数字取整为整数。因此，需要使用 Math 命名空间的 floor 方法，如下所示：

```
>Math.floor(3.2726867394521832)
3
>Math.floor(9.840337357949466)
9
```

使用 floor 方法，无论小数部分是什么，总是向下舍入。例如 10.00001、10.2、10.5、10.8 和 10.99999 都向下舍入为 10。下面，我们使用 floor 生成一个表达式，返回一个 1 到 10 之间的随机整数（包括 1 和 10）：

```
var number = Math.random() * 10 + 1 // 1<= number<11
var number = Math.floor(Math.random() * 10 + 1) // 1<= number<= 10
```

与以上 floor 方法相对应，Math.ceil 方法总是向上舍入。另外，还有 Math.round 方法，可以向上或向下舍入，二者都遵循通常的数学舍入规则。有关 JavaScript 中 Math 对象的更多内容，请参阅本书网站 http://www.room51.co.uk/js/math.html。

代码清单 12-4 将 Math 方法付诸实践。现在 guess 函数返回字符串而不输出这些字符串，控制台自动显示返回值，交互如下：

```
>guess(2)
Unlucky, try again.
>guess(8)
Unlucky, try again.
>guess(7)
Well done!
```



代码清单 12-4　猜随机数
(http://jsbin.com/mezowa/edit?js,console)

```
var getGuesser = function () {
   var secret = Math.floor(Math.random() * 10 + 1);
               //使用 Math.random 和 Math.floor 生成 1 到 10 之间的整数
   return function (userNumber) {
      if(userNumber === secret){   //仅当条件为 true 时，才使用 if 语句执行命令
            return "Well done!";  //当调用函数时，返回值将显示在控制台上
      } else {       //包含一个 else 子句，如果条件为 false，则执行该命令
            return "Unlucky, try again.";
      }
   };
};
var guess = getGuesser();       //调用 getGuesser 函数并将返回函数分配给 guess 变量
```

205

使用随机数使这款猜密码游戏变得更加有趣，但是该游戏目前还只是简单直接地猜测，没有涉及任何猜谜策略。下面，我们将进一步改进这款游戏，在用户每次猜测后都为其提供反馈。

12.3 使用 else if 创造更多条件

如果用户在每次猜测后都能得到反馈，他们就可以在玩游戏的过程中不断改进，找到更有效的猜谜策略。通常来讲，策略游戏比猜密码游戏更加有趣。如果用户的猜测不正确，请告诉他们是猜得太高还是猜得太低了，如下所示：

```
>guess(2)
Too low!
>guess(7)
Too high!
>guess(5)
Well done!
```

图 12-3 显示了针对三种用户反馈的不同条件。

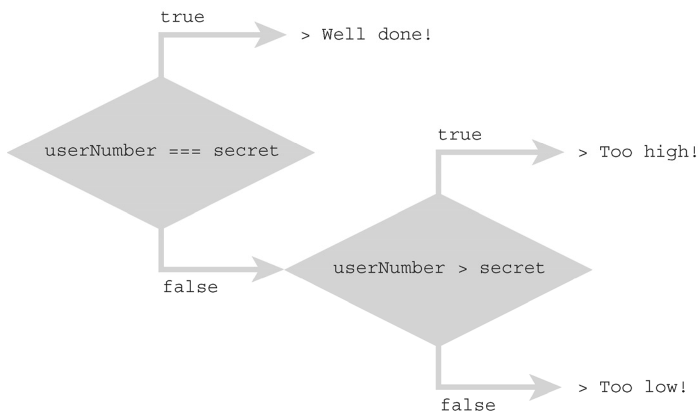


图 12-3　条件嵌套可以提供多个选项

在以下代码清单 12-5 中，使用了两个 if 语句来区分两种类型的错误答案。



代码清单 12-5　用户的猜测太高或太低 (http://jsbin.com/cixeju/edit?js,console)

206

```
var getGuesser = function () {
   var secret = Math.floor(Math.random() * 10 + 1);
   return function (userNumber) {
      if (userNumber === secret){ //使用条件来判断用户是否猜到了密码
         return "Well done!";
      } else {                    //如果猜测不正确，则执行第一个 else 子句
      if (userNumber>secret) {    //检查用户猜测的密码是否大于密码数字
         return "Too high!";
```

```
            } else {                    //用户猜测的密码不大于密码数字
                return "Too low!";
            }
        }
    };
};
var guess = getGuesser();
```

如果 if 语句花括号内的代码块只包含一条语句，这个花括号可以省略。请看以下三个代码段，三者是等效的：

```
if (userNumber === secret) {
    return "Well done!";
}
if (userNumber === secret)
    return "Well done!";
if (userName === secret)     return "Well done!";
```

伴随代码的与时俱进和不断更新，if 和 else 子句会变得越来越复杂。这时，如果省略花括号，就可能很难识别哪些语句与哪些子句是一起配套使用的。许多程序员，也包括我本人，建议读者还是不要轻易省略花括号（除了嵌套的 if 语句，稍后讲解）。当然，也有一些程序员并不这么严格要求，因此是否省略花括号可以归结为个人（或团队）喜好。本书采用读者最容易理解的方式来使用花括号。

针对每一条 if 语句，即使只带有一个 else 子句，也会被当作一条语句。当 else 子句包含单个 if 语句时，通常会省略花括号。以下三个代码段是等效的：

首先，如代码清单 12-5 所示，嵌套的 if-else 语句位于一对花括号内，如下所示：

```
else {  // 起始花括号
    if (userNumber>secret) {
        return "Too high!";
    } else {
        return "Too low!";
    }
}        // 结束花括号
```

其次，内部的 if-else 是单个语句，因此不需要用花括号，如下所示：

```
else    // 没有花括号
    if (userNumber>secret) {
        return "Too high!";
        } else {
        return "Too low!";
    }
// 没有花括号
```

207

最后，因为 JavaScript 基本上忽略空格符和制表符，所以内部的 if-else 语句可以移动到初始的 else 子句后面，如下所示：

```
else if (userNumber>secret) {     // if 移动到 else 的后面
    return "Too high!";
} else {
    return "Too low!";
}
```

在上述三个代码段中，最后一种版本是最常见的格式。以下代码清单 12-6 显示了上文中重新排序的 else-if 代码块。



代码清单 12-6　重新排序的 else-if 代码块
(http://jsbin.com/cidoru/edit?js,console)

```
var getGuesser = function () {
    var secret = Math.floor(Math.random() * 10 + 1);
    return function (userNumber) {
        if (userNumber === secret) {
            return "Well done!";
        } else if (userNumber>secret) {
                return "Too high!";
            } else {
                return "Too low!";
            }
    };
};
var guess = getGuesser();
```

为了与代码清单 12-5 进行比较，代码清单 12-6 中的第二个 if 语句用粗体显示。程序员删除了第一个 else 代码块的花括号，并将第二个 if 语句移动到第一个 else 的旁边。代码清单 12-6 是编写 else-if 块的最常用方法。 但是如果读者个人偏爱代码清单 12-5 中那个较长的版本，当然可以坚持自己的选择，没有哪个代码法官可以宣判你滥用语法罪。

在上述猜密码数字的游戏中，所有可能的输出结果只有三种：正确、太高或太低。 也就是说，如果用户的猜测不正确，并且也不是太高，那么一定是太低。

12.3.1　比较运算符

以上代码清单 12-5 和 12-6 中都使用了大于运算符>，该运算符用于比较两个值，其运算
208 结果是 true 或 false。表 12-1 列出一些用于比较两个值的运算符，大于运算符只是其中之一。

表 12-1　比较运算符

操作符	名称	举例	运算结果
>	大于	5>3 3>10 7>7	true false false
>=	大于或等于	5>= 3 3>= 10 7>= 7	true false true
<	小于	5<3 3<10 7<7	false true false

（续）

操作符	名称	举例	运算结果
<=	小于或等于	5<= 3 3<= 10 7<= 7	false true true
===	严格相等	5 === 3 7 === 7 7 === "7"	false true false
!==	不严格相等	5 ! == 3 7 ! == 7 7 ! == "7"	true false true

表 12-1 中所有运算符的运算结果都是 true 或 false，所以可以在 if 语句的条件中使用这些运算符。读者也许想知道“严格相等运算符”中的“严格”到底意味着什么，是否还有一个与之对应的“非严格”版本。这种疑惑不无道理，对于“非严格”版本的相等，程序中可以使用“==”运算符，请参阅以下“普通相等和强制转换”补充解释栏。

普通相等和强制转换

程序允许**普通相等运算符**==将两个值强制转换为相同的类型，目的是能够对这两个值进行比较，得到二者相等的结果。

强制转换是将值从一种类型转换为另一种类型的过程，例如，从字符串转换为数字。

因此，严格比较 7 === “7” 的结果为 false，因为一个值是数字，而另一个值是字符串。普通比较 7 == “7” 的计算结果为 true，因为字符串首先被强制转换为数字，然后 7 == 7 的结果就是 true。

强制转换的规则超出了本书的范围（尽管有人认为它们根本不值得一学），我们在本书中将坚持运用严格比较。

显然，经过改进的猜数字密码游戏已经变得非常有趣。下面我们继续改进以前设计的那款测验应用程序，为其增加一个检查答案的功能。 209

12.4　在测验应用程序中检查答案

既然可以通过 if 语句中的条件来判断对错，在测验结束时就可以输出用户在该测验程序中答对问题的分数，在控制台上的交互如下所示：

```
>quiz.quizMe()
What is the highest mountain in the world?
>quiz.submit("Qomolangma")
Correct!
Your score is 1 out of 1
>quiz.quizMe()
What is the highest mountain in Scotland?
>quiz.submit("Snowdon")
No, the answer is Ben Nevis
You have finished the quiz
Your score is 1 out of 2
```

该测验程序的代码在以下代码清单 12-7 中显示。getQuiz 函数包含了该测验的程序

实现，并返回一个带有 quizMe 和 submit 方法的接口对象，请读者查看该代码清单的运行情况。



代码清单 12-7 检查测验的答案
(http://jsbin.com/hidogo/edit?js,console)

```
var getQuiz = function () {
    var score = 0,    //使用单个 var 关键字声明 getQuiz 函数中的所有变量
    qIndex = 0,
    inPlay = true,
    questions,
    next,
    getQuestion,
    checkAnswer,
    submit;
    questions = [
        {
            question: "What is the highest mountain in the world?",
            answer: "Qomolangma"
        },
        {
            question: "What is the highest mountain in Scotland?",
            answer: "Ben Nevis"
        }
    ];
    getQuestion = function () {   //定义 getQuestion 方法以返回当前提问的问题
        if (inPlay) {
            return questions[qIndex].question;
        } else {
            return "You have finished the quiz.";
        }
210 };
    next = function() {   //定义 next 方法来移动到下一个问题，并检查问题是否结束
        qIndex = qIndex + 1;
        if (qIndex>= questions.length) {
            inPlay = false;
            console.log("You have finished the quiz.");
        }
    };
    checkAnswer = function (userAnswer){ //定义 checkAnswer 方法以检查答案
                                         //是否正确并更新分数
        if (userAnswer === questions[qIndex].answer) {
            console.log("Correct!");
            score = score + 1;
        } else {
            console.log("No, the answer is " + questions[qIndex].answer);
        }
```

```
        };
        submit = function (userAnswer) {  //定义 submit 方法来处理用户提交的答案
            var message = "You have finished the quiz.";
            if (inPlay) {
                checkAnswer(userAnswer);
                next();
                message = "Your score is " + score + " outof " + qIndex;
            }
            return message;
        };
        return {              //返回带有 quizMe 和 submit 方法的接口对象
            quizMe: getQuestion,
            submit: submit
        };
    };
    var quiz = getQuiz();    //调用 getQuiz 函数并将返回的接口对象分配给 quiz 变量
```

这个新的测验程序由多个可拆分部件组成，下面将对其进行分解。

12.4.1　变量的便捷声明方式：使用 var 关键字声明多个变量

在本节内容之前，我们每次声明一个变量都要使用一次 var 关键字：

```
var score;
var getQuestion;
var next;
var submit;
```

211

其实，JavaScript 允许使用一个 var 关键字同时声明多个变量，各个变量之间用逗号分隔，最后用分号来结束。因此，上述 4 个变量声明可以改写为以下形式：

```
var score,
getQuestion,
next,
submit;
```

还可以将需要声明的变量写在同一行代码中，如下所示：

```
var score, getQuestion, next, submit;
```

不过，大多数程序员喜欢每行只写一个变量。另外，在使用一个 var 关键字声明多个变量的过程中还可以加入赋值：

```
var   score = 0,
getQuestion,
next,
submit = function (userAnswer) {
    // 函数体
};
```

这种便捷的变量声明方式，目的在于确保所有变量都得以声明，而且代码更易于阅读和理解。我本人偏爱代码清单 12-7 中的风格，感觉它更易于阅读，而且还能少打一些字。有些程序员确实喜欢在声明每一个变量时都使用 var 关键字，且单独一行，就像本书前面章节中一样，这种做法为将来剪切和粘贴变量提供了方便。其实我们没必要操心别人的做法有什么优点，随着时间的推移，每个程序员都会逐渐形成自己的风格。

12.4.2 显示测验中的问题

在以上代码清单 12-7 中，getQuestion 函数从 questions 数组中获得一个测验问题，使用 qIndex 变量从数组中选取当前的问答对象，并返回问答对象的 question 属性，如下所示：

```
return questions[qIndex].question;
```

只有在测验进行过程中，该函数才会返回测验问题，否则，将返回一个字符串来说明测验结束，如下所示：

```
return "You have finished the quiz.";
```

该程序使用 inPlay 变量来标记测验正在进行中还是已经完成。当测验正在进行时，inPlay 变量的值为 true；当测验结束时该变量的值为 false。getQuestion 函数将 inPlay 变量用于 if 语句中的条件：

212

```
if (inPlay) {
   return questions[qIndex].question;
} else {
   return "You have finished the quiz.";
}
```

当 inPlay 为 true 时，函数返回的是测验的问题；当 inPlay 为 false 时，函数返回的是字符串消息（请注意：在控制台上的提示符处调用一个函数时，控制台会自动显示返回值）。

12.4.3 移至下一道题目

在以上代码清单 12-7 中，调用 next 函数，从一道题目移动到下一道题目，具体实现方式是变量 qIndex 的递增，如下所示：

```
qIndex = qIndex + 1;
```

该程序将 questions 数组中当前元素的索引值存储在变量 qIndex 中。请注意，数组索引值是从零开始的，因此对于长度为 4 的数组，索引值分别为 0，1，2，3。索引值如果为 4，就会超出数组的末尾（3 是该数组的最后一个索引值）。一般来说，如果索引值大于或等于数组的长度，就会超过数组的末尾。所有的数组都有一个 length 属性。在测验程序中，length 属性代表问题的数量。

在以下语句中，next 函数在检查当前元素的索引值是否超过了最后一个问题的索引值：

```
if (qIndex >= questions.length)
```

如果索引值超过了数组的末尾，说明所有的问题都回答完毕，就结束测验。因此，将

inPlay 赋值为 false，如下所示：

```
if (qIndex>= questions.length) {
   inPlay = false;
   console.log("You have finished the quiz.");
}
```

12.4.4　检查用户的答案

在以上代码清单 12-7 中，使用了 checkAnswer 函数。如果用户提交的答案与来自问题数组的当前答案完全相同，玩家的得分就会增加 1 分，否则，显示正确的答案，如下所示：

```
if (userAnswer === questions[qIndex].answer) {
   console.log("Correct!");
   score = score + 1;
} else {
   console.log("No, the answer is " + questions[qIndex].answer);
}
```

213

12.4.5　处理用户的答案

在以上代码清单 12-7 中，当玩家提交答案后，用 submit 函数来编排程序的后续进程，该函数的返回值为以下两种情况之一：用户的得分或者提示测验已结束的消息，如下所示：

```
Your score is 1 out of 2      // 如果 inPlay 的值是 true
You have finished the quiz.   //如果 inPlay 的值是 false
```

如果测验仍在进行当中，submit 将调用另外两个函数：checkAnswer 和 next，每个函数都将按顺序执行其代码。我们可以看到，程序正在按照需要依次运行代码，如下所示：

```
if (inPlay) {
   checkAnswer(userAnswer);
   next();
   message = "Your score is " + score + " outof " + qIndex;
}
```

12.4.6　返回接口对象

在以上代码清单 12-7 中，getQuiz 返回的接口对象比较简单，它并没有自己的实现代码，而是从 getQuiz 的局部作用域内得到两个函数的赋值作为其属性，如下所示：

```
return {
   quizMe: getQuestion,
   submit: submit
};
```

正如第 11 章所述，接口对象有助于保持一个始终不变的接口，即使 getQuiz 中的功能实

现被更改，用户依然可以调用 quiz.quizMe()和 quiz.submit()。用户可以更改以上赋值给接口对象属性的两个函数以及这些函数的工作方式，但是千万不要删除或重新命名这两个属性。

请读者仔细体会程序如何通过代码片段进行分工，这些片段又是如何协同工作来实现程序的整体功能。程序员在编写程序时，有一个贯穿始终的目标：代码容易阅读，并且容易理解。if 语句及其 else 子句确实有助于程序员实现上述目标，使用这些语句可以清晰地梳理程序的流程，明确程序在每个节点所采取的适当操作。

下面，我们将把这些新的概念应用到游戏 The Crypt 中去。

12.5 The Crypt——检查用户的输入

在第 11 章中，我们创建了一个 getGame 函数，为 The Crypt 返回一个公共接口。玩家通过调用从 getGame 返回的 go 方法就可以从一个地方移动到另一个地方，通过调用从 getGame 返回的 get 方法就可以拾取物品，如下所示：

214

```
return {
    go: function (direction) {
        var place = player.getPlace();
        var destination = place.getExit(direction);
        player.setPlace(destination);
        render();
        return "";
    },
    get: function () {
        var place = player.getPlace();
        var item = place.getLastItem();
        player.addItem(item);
        render();
        return "";
    }
};
```

12.5.1 go 方法解读

下面，我们逐条分析以上 go 方法中的三行语句，请读者寻找其中潜在的问题。

1. 获取玩家的位置

首先，从 getPlace 方法开始。我们使用 getPlace 方法返回玩家的当前位置：

```
var place = player.getPlace();
```

该方法将当前位置分配给 place 变量，如果玩家目前在厨房，那么以上代码就等同于：

```
var place = kitchen;
```

该程序使用 setPlace 方法预先设置了玩家的初始位置，如下所示：

```
player.setPlace(kitchen);
```

2. 通过方向来寻找目的地

既然程序已经获得了当前位置，就可以通过调用该位置的 getExit 方法，利用指定的方向来检索目的地，如下所示：

```
var destination = place.getExit(direction);
```

当玩家调用 go 方法时，将实参传递给 direction 形参。

```
>game.go("south")
```

以上命令等同于以下代码：

```
var destination = place.getExit("south");
```

如果图书馆在厨房的南边，那么该代码相当于：

```
var destination = library;
```

215

3. 将玩家移动到目的地

既然程序已经找到了目的地，下一步只需要更新玩家的位置，如下所示：

```
player.setPlace(destination);
```

太好了！在游戏中，用户可以自由决定去哪里了。但问题是，我们是否允许用户在程序员精心制作的城堡里自由行走？不行！我们知道有些用户是有坏心思的，他们会趁虚而入！

12.5.2　永远不能轻信用户输入的信息

抱歉，也许是我有些过虑了。当然用户本身并不邪恶，但他们在输入信息时难免会有一些手误，有些用户还会养猫，这些猫大多不老实，常常在键盘上动手动脚，而且它们又不会输入，因此错误更是在所难免。当程序期望用户提供输入时，我们必须提防用户输入中的各种错误，无论是输入手误（可能是猫引起的），还是出于好奇对程序内部的窥探。

go 方法希望用户以字符串的形式输入一个有效方向，并利用该方向寻找目的地，即玩家想要去的地方。但是，如果用户输入了一个不存在的方向，如下所示，整个游戏就会中断。

```
>game.go("snarf")
```

在 JS Bin 上运行 The Crypt 中的第 11 章的版本，当输入以上命令后，控制台上就会出现以下报错消息，如图 12-4 所示。（http://jsbin.com/yuporu/edit?js,console）

```
> game.go("snarf")
✖ "undefined is not an object (evaluating 'place.showInfo')"
> game.go("south")
✖ "undefined is not an object (evaluating 'place.getExit')"
> |
```

图 12-4　输入一个不存在的方向导致游戏中断

使用不同的浏览器，以上报错消息可能会略有不同。在报错消息之后，即使再输入一个有效的方向，也不能解决问题。从图 12-4 中的报错消息来看，问题可能出在 place 变量，go 方法的关键语句中使用了用户输入的信息：

```
var destination = place.getExit(direction);
```

如果指定方向并不是该位置的出口，那么 getexit 函数将返回 undefined。该程序把 undefined 分配给 destination 变量，并将该值设置为玩家的新位置：

216
```
player.setPlace(destination);
```

现在玩家的位置是 undefined，而不是使用 Place 构造函数生成的地方。undefined 这个地方根本没有 showInfo 或 getExit 方法，压根儿没有任何方法！现在，读者应该对图 12-4 中的错误有了更深入的理解。

那么，我们应该如何防范用户（和他们的猫）犯错误？

12.5.3 确保探索不会中断——使用 if 语句

为了避免出现上述问题，在更新玩家的位置之前，我们可以使用 if 语句来检查目的地是否有效：

```
go: function (direction) {
    var place = player.getPlace();
    var destination = place.getExit(direction);
        if (destination !== undefined) {
            player.setPlace(destination);
            render();
            return "";
        }else {
            return "*** There is no exit in that direction ***";
    }
}
```

如果在当前位置并没有指定方向的出口，则 getExit 方法返回 undefined。 在调用 setPlace 之前，只需要检查 destination 是否为 undefined，如下所示：

```
if (destination !== undefined) {
    //目的地有效
}
```

在表 12-1 中我们可以看到，当两个值不相等时，!==运算符返回 true；当二者相等时，返回 false。可以添加一个 else 子句来应对目的地确实未定义的情况，如下所示：

```
if (destination !== undefined) {
    //目的地有效
} else {
    // 该指定方向无出口
}
```

以下代码清单 12-8 是从 getGame 函数返回的 go 方法和 get 方法的更新版本。在控制台上输入不存在的方向，就会出现如下交互：

```
>game.go("snarf")
  *** You can't go in that direction ***
```

如果没有任何东西可以捡拾，调用 get 方法，会出现如下交互：

```
>game.get()
  *** There is no item to get ***
```

217

在此代码清单中，只显示了更新部分的代码，完整列表在 JS Bin 上可见。



代码清单 12-8　检查用户的输入 (http://jsbin.com/zoruxu/edit?js,console)

```
var getGame = function () {
    var spacer = { … };
    var Player = function (name, health) { … };
    var Place = function (title, description) { … };
    var render = function () { … };
    var kitchen = new Place("The Kitchen", "You are in a kitchen…");
    var library = new Place("The Old Library", "You are in a library…");
    kitchen.addExit("south", library);
    library.addExit("north", kitchen);
    // 游戏初始化
    var player = new Player("Kandra", 50);
    player.addItem("The Sword of Doom");
    player.setPlace(kitchen);
    render();
    return {          //返回一个带有 go 和 get 方法的接口对象
        go: function (direction) {
            var place = player.getPlace();
            var destination = place.getExit(direction);
                                   //使用 getExit 找到用户指定方向的目的地
            if (destination!==undefined){ //检查目的地是否未定义，即检
                                          //查它是否有效
                player.setPlace(destination);     //只有在目的地存在时才设置和
                                                  //显示新位置
                render();
                return "";
            } else { //添加一个 else 子句，用于处理目的地是 undefined 的情况
                return "*** You can't go in that direction ***";
                            //向用户反馈指定的方向无效
            }
        },
        get: function () {    //更新 get 方法，对其进行与 go 方法类似的检查
            var place = player.getPlace();
```

```
      var item = place.getLastItem();
      if (item !== undefined) {
        player.addItem(item);
      render();
      return "";
      } else {
        return "*** There is no item to get ***";
      }
    }
  };
};
var game = getGame();
```

218

在以上代码清单 12-8 中，省略了 Player 和 Place 函数构造的细节，以便将读者的关注点聚焦在对 go 和 get 方法的改进。在第 13 章中，读者将学习如何把每个函数的构造放在该函数自己的文件中，还将学习如何在 JS Bin 中导入文件。这种逐渐增强的模块化概念有助于程序员每次只关注一个焦点，并且有助于程序员在多个项目中重用某些代码。

12.6 本章小结

■ 使用比较运算符比较两个值，运算结果为返回布尔值 true 或 false：

```
> 5 === 5     // 严格相等
    true
> 10 > 13     // 大于
    false
```

■ 仅当条件满足时，才使用 if 语句执行代码：

```
if (condition) {
    // 如果条件的值为 true，执行代码
}
```

■ 使用比较运算符和（或）变量设置条件：

```
if (userNumber === secret) {
    // 如果 userNumber 和 secret 完全相等，执行代码
}
if (inPlay) {
    // 如果 inPlay 的值为 true，执行代码
}
```

■ 当条件不满足时，添加 else 子句执行代码：

```
if (condition) {
    //如果条件的值为 true，执行代码
} else {
```

```
    //如果条件的值为 false，执行代码
}
```

- 在 else 子句中包含额外的 if 语句来处理更多的可能性：

```
if (userNumber === secret) {
    console.log("Well done!");
} elseif (userNumber>secret) {
    console.log("Too high!");
} else {
    console.log("Too low!");
}
```
219

- 使用 Math.random( )生成随机数，生成的数字在 0 和 1 之间，可以等于 0，但不可以等于 1：

```
>Math.random()
  0.552000432042405
```

- 将随机数按比例放大到所需的范围：

```
Math.random() * 10                    // 0<= decimal<10
Math.random() * 10 + 1                // 1<= decimal<11
```

- 使用 Math.floor( )将随机数舍入为整数：

```
Math.floor(Math.random() * 10 + 1)     // 1<= integer<= 10
```

- 不能相信用户输入的数据，要对其进行检查，以确保输入有效。 220

第 13 章　模块：分解程序

本章内容包括：

- 使用 script 元素将代码导入 JS Bin
- 避免重复的变量名
- 运行未分配变量的函数
- 使用模块组织（共享）代码库

随着我们开发的应用程序越来越大，涉及的变量、对象、数组和函数也会越来越多，因此在单个程序文件内高效地运行代码也会变得越来越困难。好的文本编辑器和开发环境会对开发者提供一些帮助，但即使拥有这些工具，也需要想办法将代码拆分成多个文件。

例如，在游戏 The Crypt 中就有 spacer、玩家、位置、地图和游戏自身逻辑等各个元素。读者可能已经注意到，包括以上所有元素的代码清单在 JS Bin 上已经变得相当长了。如果能够将每个元素的代码放在不同的文件中，就能够帮助读者每次只关注一段程序，还能够帮助各个程序员更加容易地开发和测试应用程序的不同部分。图 13-1 显示如何将这个大程序分解成为各个小模块。

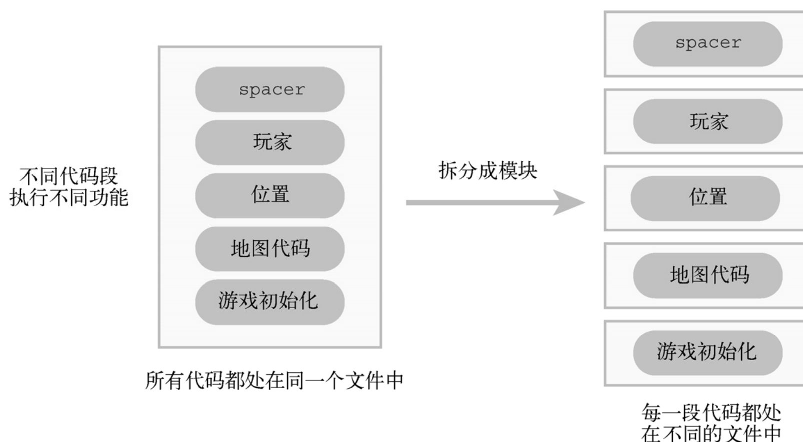


图 13-1　将一个大程序分解成为各个小模块

在不同文件中使用具有独立功能的代码片段和数据也有助于代码重用。开发者不再需要将 JavaScript 的某些有用的函数和代码片段从一个项目剪切和粘贴到其他项目中，而是可以将其保留在某个单个的库文件中，在需要时将其导入到其他项目中。例如，第 7 章中的 spacer 命名空间，用于在控制台上格式化文本，开发者在测验程序和博客程序中都可以使用该命名空间，开发者希望不必在每个应用程序中都重复 spacer 代码，而是将其放在它自

己的文件中，在需要时导入该文件即可（图 13-2）。

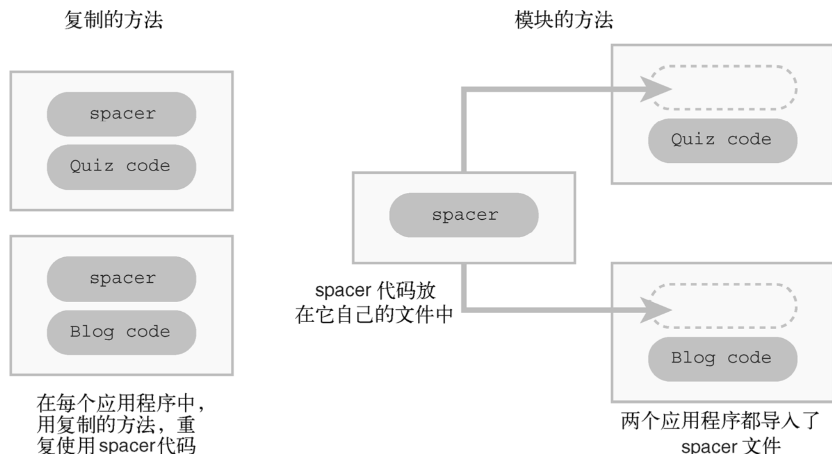


图 13-2　将 spacer 移动到一个模块中，这样就可以在多个项目中使用该模块 222

在本书中，将这些具有某个特定功能的文件统称为模块。目前，业界有许多已发布的 JavaScript 项目和用于管理模块的标准，在这些标准中，对模块必须采用的形式有更严格的定义，最新版本的 JavaScript 正在推出自身的模块系统。它们对模块的定义严谨而复杂，在本书中，就暂且简单粗略地使用模块概念了。

本章讨论如何使用 HTML script 元素将模块导入 JS Bin，读者将看到如何把随机数生成函数和 spacer 命名空间的文本格式化功能合并到其他项目中去。在着手将各个模块的代码植入到某单个程序中时，请务必密切关注正在使用哪些变量名称，因为植入过程中存在风险，即导入的文件有可能会覆盖程序中已有的变量。为了规避以上风险，我们引入以下两种方法：使用命名空间和使用立即调用函数表达式。立即调用函数表达式是一种不将函数分配给变量而直接运行函数代码的方法。

下面，我们先来了解 JS Bin 如何处理文件。

13.1　了解 JS Bin 上的 bin 和文件

在本书第 12 章，我们创建了一个简单的游戏，让玩家在 1 到 10 之间猜测一个数字，该数字由 Math.random 方法生成。现在，我们想采用类似的方式来更新测验应用程序，从问题库中随机显示一个问题。以上猜数字游戏和测验程序都需要在两个界限范围之内生成一个随机整数，因此，二者都可以使用以下函数：

```
var between = function (lowest, highest) {
    //返回一个介于最低值和最高值之间的整数
};
```

在本节中，读者将学习如何在 JS Bin 上创建并保存包含数字生成器代码的 JavaScript 文件；在 13.2 节中，读者将学习如何加载文件；在 13.3 节中，读者将学习如何将该文件加载到猜数字游戏和测验应用程序中，如图 13-3 所示。

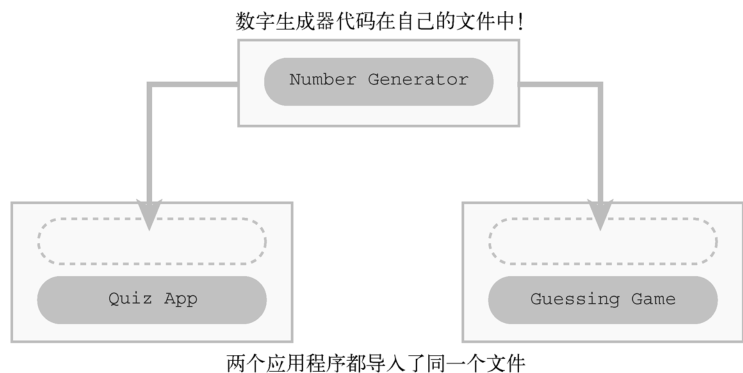


图 13-3　测验应用程序和猜数字游戏都导入了数字生成器代码

223 在学习如何导入文件之前，读者需要简单了解 JS Bin 如何保存文件。JS Bin 是一个简单的开发环境，允许用户在各自独立的面板中处理 HTML、CSS 和 JavaScript 代码。图 13-4 显示了这三个面板同时打开的界面，在每个界面中都有一些代码。



图 13-4　JS Bin 上的 HTML、CSS 和 JavaScript 面板

JS Bin 可以将 HTML、CSS 和 JavaScript 面板中的代码合并生成一个我们所需要的网页，并显示在输出（Output）面板中（图 13-5）。在本书的第三部分将更加详细地介绍 HTML 和 CSS 的相关知识。



图 13-5　HTML、CSS 和 JavaScript 代码合并生成网页输出

代码运行过程中发生的任何错误或警告都会在控制台（Console）面板中显示，该面板还可以显示 JavaScript 代码的输出——这是迄今为止我们在本书中一直采用的显示输出方

式。读者目前一直在专心学习 JavaScript，因此不必分心去考虑如何根据 Output 面板中的内容生成网页。打开链接 http://jsbin.com/jejunu/edit?output，即可在 JS Bin 中查看图 13-4 和 13-5 中的页面，在不同的面板之间切换可以查看代码。 224

JS Bin 不仅为程序员提供一个环境来编辑合成构建网页的各种类型代码，还可以将 HTML、CSS 和 JavaScript 代码作为单独的文件进行访问。想要创建 JavaScript 代码并将其视为单独的文件，请在 JS Bin 中按照以下步骤进行操作：

（1）创建一个 bin。

（2）在 JavaScript 面板中编写一些代码。

（3）记录文件名。

（4）查看单个代码文件。

13.1.1 创建一个 bin

在 JS Bin 的 file 菜单上，单击 new，就可以创建 HTML、CSS 和 JavaScript 三个文件，并在相应面板上显示每个文件的内容，这三个文件统称为一个 bin。对于一个新创建的 bin，其 HTML 文件中已经包含了大多数新网页通用的样板代码，而 CSS 文件和 JavaScript 文件中则是空白的。

13.1.2 编写代码

请将以下代码添加到 JavaScript 面板中：

```
var between = function (lowest, highest) {
   var range = highest - lowest + 1;
   return lowest + Math.floor(Math.random() * range);
};
```

在以上代码中，between 函数将返回一个在 lowest 和 highest 之间的随机整数。例如，between(3, 5)将返回 3、4 或 5。

13.1.3 记录文件名

JS Bin 为每个 bin 分配一个代码，用于编辑和访问 bin 内的文件。请查看浏览器地址栏中的当前网址（有可能需要单击地址栏，才能看到完整的地址）。图 13-6 展示了这样一个网址，其 bin 代码为 qezoce，可见面板为 HTML、JavaScript 和 Console。



图 13-6 JS Bin 网址（URL）分解图

请读者记录一下自己当前工作的 bin 代码，它肯定不同于图 13-6 所示的 bin 代码，因为 qezoce 是本书作者当前工作的 bin 代码。 225

13.1.4 查看单个代码文件

要访问某个代码文件，请使用不同的 URL 格式，如图 13-7 所示。该格式包含一个 output 前缀，以 bin 代码和文件扩展名结尾。该文件扩展名指定要加载的文件类型，js 代表 JavaScript。



图 13-7 一个 JavaScript 文件的 JS Bin URL

访问 http://output.jsbin.com/qezoce.js，只是加载 JavaScript 文件，如图 13-8 所示，并不会加载 JS Bin 编辑环境的所有面板、菜单及控件，该文件只是纯文本内容的 JavaScript 文件。



图 13-8 JS Bin 上的 JavaScript 文件

在该文件中，可能已经删除了不必要的空格和换行符，因此在我们人类看来，就感觉格式不够整齐和美观。可以试试在 JS Bin 上使用 bin 代码加载文件。非常好！我们已经得到了一个文件的纯 JavaScript 代码，下一步就是如何将它植入到其他程序中。

13.2 将文件导入其他项目

我们将使用上一节的数字生成函数 between 来创建一个程序，步骤如下：

（1）创建一个 bin。

（2）在 JavaScript 面板中编写代码。

（3）将 script 元素添加到 HTML 面板。

（4）刷新页面。

226 （5）运行程序。

13.2.1 创建一个 bin

在 JS Bin 上，单击 file 菜单上的 new，就可以创建一个 bin。系统会自动重置 HTML，CSS 和 JavaScript 面板。

13.2.2 编写代码

在 JavaScript 面板中输入以下代码：

```
// 需要数字生成器模块
var num = between(3, 7);
console.log(num);
```

如果运行以上程序，就会产生一个错误：缺少对 between 的变量声明或函数定义。因为 between 函数处在一个单独的文件中，这样就引发了依赖问题。当程序员将代码拆分成模块后，一个模块需要依赖另一个模块才能够运行的情况并不罕见，以上代码就依赖于数字生成器模块才能够运行。更高级的模块系统通常会自动加载各个依赖关系，并让程序员明确地记录这些依赖关系。本书中并没有采用这些高级的模块系统，因此采用添加注释的方式来显示依赖关系所需要的模块。

13.2.3　添加 script 元素

现在，我们开始使用 JS Bin 上的 HTML 面板。HTML 是用于构成网页结构和内容的代码，用于指定标题、段落、列表以及链接等内容，详细内容将在第 17 章中进行介绍，并在本书第三部分得以充分运用。当前，读者的学习重点依然是 JavaScript，JS Bin 用来帮助读者学习和探索 JavaScript。我们只是使用少许的 HTML 片段，将很长的 JavaScript 程序分解成为单独的文件，并根据需要加载各文件。我们使用 HTML script 元素指定要加载的 JavaScript 文件。图 13-9 显示了构成 HTML script 元素的各个部分。

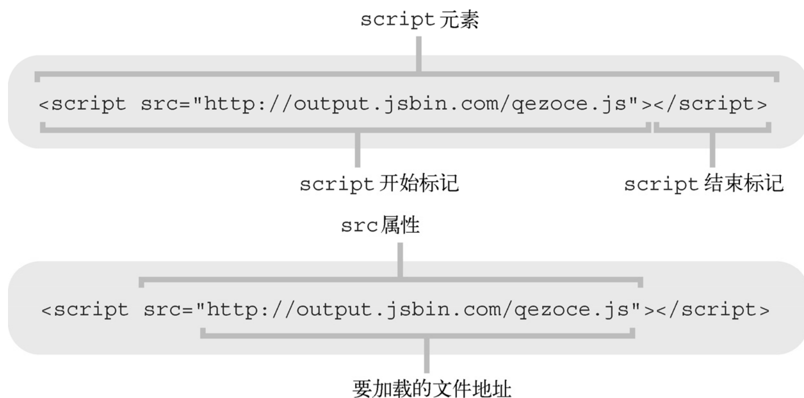


图 13-9　script 元素的构成 227

读者不必担心上述 HTML 的各种术语，也不需要完全了解 HTML 的元素、标记和属性，只要学会如何使用 script 元素加载文件即可。关于 HTML 的更多内容，将在第 17 章中继续讲解。

在 JS Bin 上显示 HTML 面板，就会看到一些默认的 HTML 代码。谨记，我们现在的目标并不是创建一个网页，而是要加载一个 JavaScript 文件。请使用在上一节中创建文件的 bin 代码替换默认的 HTML 代码来加载 JavaScript，如下所示：

```
<script src="http://output.jsbin.com/qezoce.js"></script>
```

被包含的 HTML 是具有 src 属性的单个 script 元素。程序员可以使用 script 元素来加载 src 属性指定的 JavaScript 文件（src 是 source 的缩写形式，指明文件的地址）。通常，要用下面的格式来加载文件：

```
<script src="path/to/someFile.js"></script>
```

新版本的浏览器会默认文件已经包含了 JavaScript，但对于老版本的浏览器，程序员还是要手工添加一个 type 属性。

```
<script src="path/to/someFile.js" type="text/javascript"></script>
```

13.2.4 刷新页面

JS Bin 并不能完全自动加载文件，可能需要在添加 script 元素后刷新浏览器页面。

13.2.5 运行程序

单击 Run 按钮，控制台面板上会显示一个介于 3 和 7 之间的数字，继续单击 Run 按钮可以生成更多的随机数。

代码清单 13-1 和 13-2 是同一程序的 HTML 和 JavaScript，两个代码清单的 JS Bin 链接指向同一个 bin。运行 5 次该程序，会产生以下输出（数字是随机的！）：

```
> 5
> 7
> 7
> 3
> 4
```



代码清单 13-1　使用 script 标签加载 JavaScript（HTML）
(http://jsbin.com/lifugam/edit?html,js,console)

228

```
<script src="http://output.jsbin.com/qezoce.js"></script>
```



代码清单 13-2　JavaScript 面板中的代码
(http://jsbin.com/lifugam/edit?html,js,console)

```
//需要数字生成器模块
var num = between(3, 7);
console.log(num);
```

运行程序时，JS Bin 首先加载并运行 script 元素的 src 属性指定的文件，然后运行 JavaScript 面板中的所有代码。被加载的文件和 JavaScript 面板中的代码一起构成了下面这个程序：

```
// 来自要加载的文件
var between = function (lowest, highest) {
   var range = highest - lowest + 1;
   return lowest + Math.floor(Math.random() * range);
};
// 来自 JavaScript 面板
var num = between(3, 7);
console.log(num);
```

当单击控制台面板中的 Run 按钮时，JS Bin 可能需要一点时间才能加载 script 标签中指

定的文件。代码运行后，将在控制台上输出随机数。

13.3　导入数字生成器——进阶示例

在本章 13.1 节，我们学习了 JS Bin 如何为程序员创建的每个 bin 分配代码，以及如何使用该代码来访问项目中的各个文件。在 13.2 节，以随机数生成器函数 between 为例，我们学习了如何创建并访问包含模块的 JavaScript 文件。程序员将工作分解为模块的目标之一就是在多个项目中能够导入和使用相同的代码，而不是复制和粘贴这些代码，如图 13-10 所示。

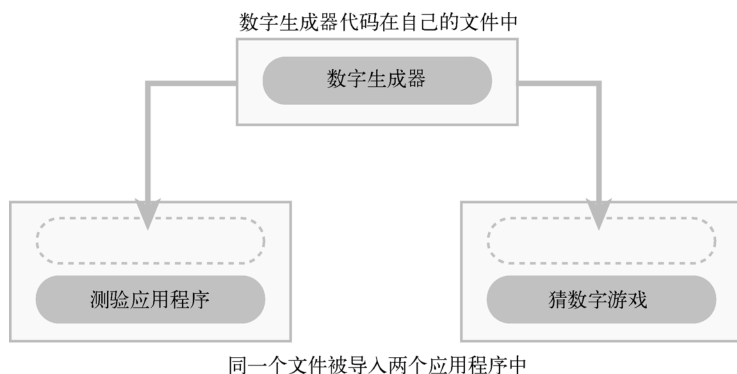


图 13-10　将数字生成器导入两个项目 229

下面，我们将数字生成器导入测验应用程序和猜数字游戏两个项目，以便更加深入地理解以上概念。

13.3.1　在测验应用程序中随机选择问题

首先，对测验应用程序进行随机化处理，每次在控制台上调用 quizMe 时，新版本程序都会从问题库中随机显示一个问题，如下所示：

```
>quiz.quizMe()
5 x 6
>quiz.submit("30")
Correct!
>quiz.quizMe()
7 x 8
>quiz.submit("30")
No, the answer is 56
```

代码清单 13-4 显示了该测验应用程序主体部分的 JavaScript。通过使用 HTML 面板上的 script 元素（代码清单 13-3）导入 between 函数的代码。



代码清单 13-3　在测验应用程序中使用数字生成器（HTML）
(http://jsbin.com/ponogi/edit?html,js,console)

```
<script src="http://output.jsbin.com/qezoce.js"></script>
```



代码清单 13-4　在测验应用程序中使用数字生成器
(http://jsbin.com/ponogi/edit?html,js,console)

```
var getQuiz = function () {    //把代码包裹在函数内以创建局部作用域
    var qIndex = 0;
    var questions = [
        { question: "7 x 8", answer: "56" },
        { question: "12 x 12", answer: "144" },
        { question: "5 x 6", answer: "30" },
        { question: "9 x 3", answer: "27" }
    ];
    var getQuestion = function () {
        qIndex = between(0, questions.length - 1);  //使用 between 函数随
                                                    //机选择一个问题
        return questions[qIndex].question;
    };
    var checkAnswer = function (userAnswer) {
        if (userAnswer === questions[qIndex].answer) {
            return "Correct!";
        } else {
            return "No, the answer is " + questions[qIndex].answer;
        }
230 };
    return {                      //从 getQuiz 函数返回一个接口对象
        quizMe: getQuestion,
        submit: checkAnswer
    };
};
var quiz = getQuiz();             //把 getQuiz 返回的接口分配给 quiz 变量
```

在以上代码清单中，between 函数从问题库中挑选了一个随机问题，questions 数组中的元素数量由 questions.length 决定（每个数组都有一个 length 属性），问题索引值是从 0 到小于 length 的数字。如果数组中有 4 个元素，那么索引值就是从 0 到 3。因此，使用下面的语句就可以选择一个随机索引：

```
qIndex = between(0, questions.length - 1);
```

程序员可以使用第 11 章中讲到的打包-返回的模块方式，把代码的功能实现（使程序正常运行的代码）隐藏在 getQuiz 函数里，然后返回一个公共接口对象。

请按照代码清单提供的链接，到 JS Bin 去测试一下你的乘法水平！所有答案都是以字符形式存储，因此，请确保你提交的是字符串，程序才能够检测，例如应提交 quiz.submit("30")，而不是 quiz.submit(30)。

13.3.2　在猜数字游戏中使用 between 函数

代码清单 13-6 显示了猜数字游戏的 JavaScript 代码。在控制台提示符下，玩家必须在 5

到 10 之间猜测数字，交互如下：

```
>guess(7)
Too high!
>guess(5)
Too low!
>guess(6)
Well done!
```

该应用程序使用了 between 函数，所以导入该函数时需要使用 HTML 面板（如清单 13-5 所示）中的 script 元素。



代码清单 13-5　在猜数字游戏中使用数字生成器(HTML)
(http://jsbin.com/tixina/edit?html,js,console)

```
<script src="http://output.jsbin.com/qezoce.js"></script>
```

231



代码清单 13-6　在猜数字游戏中使用数字生成器
(http://jsbin.com/tixina/edit?html,js,console)

```
var getGuesser = function(lowest,highest){  //把代码包裹在函数中以创建局部作用域
    var secret = between(lowest, highest);  //使用 between 函数选择一个随机数
    return function (userNumber) {          //返回一个供玩家做猜测的函数
        if (userNumber === secret) {
            return "Well done!";
        } else if (userNumber> secret) {
            return "Too high!";
        } else {
            return "Too low!";
        }
    };
};
var guess = getGuesser(5, 10);              //将返回的函数赋值给 guess 变量
```

请按照以上代码清单所提供的链接，去 JS Bin 上玩猜数字游戏！

在以上猜数字游戏和测验应用程序中都导入了相同的数字生成器文件。这种将数字生成器代码存放在一个文件中的做法保证了该数字生成器的稳定性，对其任何更新或修改只需在此单一文件上进行即可，以后所有使用该文件的项目加载的就是该文件的最新版本。

我们看到，在编程项目中文件导入是非常有用的。下面介绍如何加载多个文件。

13.4　导入多个文件

在第 7 章和第 11 章中，我们学习了如何将对象用作命名空间，只需要使用一个变量就可以将多个属性和多个方法集合在一起。例如，我们创建的 spacer 是一个函数的命名空间，用于在控制台上对文本进行格式化，该方法可以用在许多项目中。当程序员需要使用边框来格式化输出文本时，就可以使用该命名空间。如何使用？并不是将 spacer 代码复制和粘贴到

每一个需要它的程序里面。命名空间看起来更像一个替补队员，平时呆在自己的文件中待命，在需要时就挺身而出，被导入到所需程序中。请打开链接 http://jsbin.com/juneqo/edit?js 查阅 spacer 代码。

现在我们就来邀请 spacer 出山工作。在以上代码清单 13-6 中，通过导入 between 函数更新了猜数字游戏，使用该函数生成让用户猜测的密码数字。假设我们现在想把给玩家的反馈信息进行一定的格式化，也就是把反馈消息包装在一定格式的边框中，那就非 spacer 莫属了。游戏在控制台上的交互如下：

232

```
>guess(10)           >guess(10)               >guess(10)
 +++++++++++++        ------------              ==============
 +  Too high!+        - Too low! -              = well done! =
 +++++++++++++        ------------              ==============
```

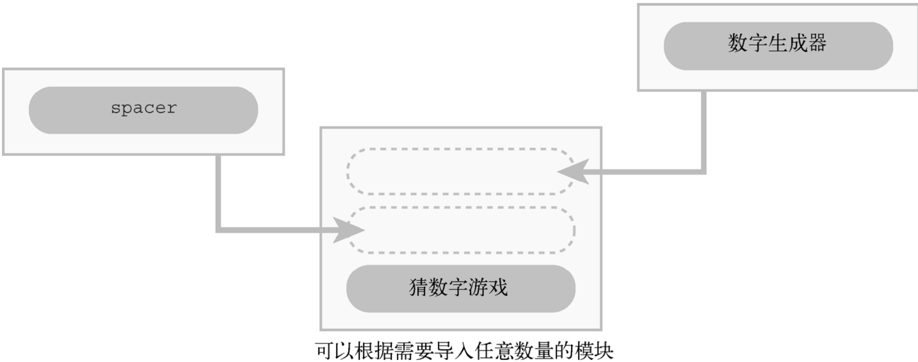


图 13-11　导入数字生成器和 spacer 模块

图 13-11 显示猜数字游戏应用程序导入了数字生成器和 spacer 两个模块。

代码清单 13-7 显示添加到 HTML 面板的 script 元素，用于导入程序正在使用的两个模块。

代码清单 13-7　在猜数字游戏中使用 spacer 和 between（HTML）(http://jsbin.com/foqowa/edit?html,js,console)

```
<!--数字生成器 -->
<script src="http://output.jsbin.com/qezoce.js"></script>
<!-- spacer 模块 -->
<script src="http://output.jsbin.com/juneqo.js"></script>
```

在上述代码中，添加注释的作用在于说明正在导入哪些模块。JS Bin 地址对用户不是非常友好，因此有必要清楚地标明正在试图加载的内容。这些注释是 HTML 注释，因此看起来与 JavaScript 注释有所不同。

以下代码清单 13-8 显示了猜数字游戏如何使用导入的 between 和 spacer 两个模块。

代码清单 13-8　在猜数字游戏中使用 spacer 和 between (http://jsbin.com/foqowa/edit?html,js,console)

```
var getGuesser = function (lowest, highest) {
```

```
        var secret = between(lowest, highest);//使用导入的between函数
        return function (userNumber) {
            if (userNumber === secret) {
                return spacer.box("Well done!",14,"=");//使用导入的spacer命名空间
            } else if (userNumber> secret) {
                return spacer.box("Too high!", 13, "+");
            } else {
                return spacer.box("Too low!", 12, "-");
            }
        };
    };
    var guess = getGuesser(5, 10);
```


在编程实践中，程序员常常将已经编写和测试过的代码存放在单独的文件中，这样做有助于他们将主要精力集中在编写新代码的工作上。代码清单 13-8 简短而精炼，归功于其中打包导入了我们信赖的 spacer 代码。

当程序员导入 JavaScript 时，就仿佛是将所有导入的代码都连接在一起，从而形成一个文件。如果在不同的导入文件中使用了相同的变量名，那么后来的代码就可能会无意中覆盖早期的代码，这样就会导致变量之间发生冲突！

13.5　冲突——新导入的代码会覆盖原先的变量

程序员感觉猜数字游戏中给用户反馈的带边框消息有些多余，现在希望反馈信息如下所示：

```
>guess(10)                    >guess(9)
 + T-o-o- -h-i-g-h-! +         = W-e-l-l- -d-o-n-e-! =
```

这些消息占用的空间更少，而且各个字符之间被破折号整齐地分隔。如何实现上述格式？我碰巧有个朋友 Kallie 一直在使用她自己的格式化函数，还把这些函数打包成了一个可以导入的模块，命名为 dasher，非常易于拿来使用：

```
dasher("message");      // m-e-s-s-a-g-e
dasher("Too low!");     // T-o-o- -l-o-w-!
```

我马上在猜数字游戏的 HTML 面板中添加一个 script 元素，来导入 Kallie 的代码，如下所示：



代码清单 13-9　导入 Kallie 的代码（HTML）
(http://jsbin.com/zusodu/edit?html,js,console)

```
<!--数字生成器-->
<script src="http://output.jsbin.com/qezoce.js"></script>
<!—spacer模块 -->
<script src="http://output.jsbin.com/juneqo.js"></script>
<!—Kallie的代码 -->
```

```
<script src="http://output.jsbin.com/soxeke.js"></script>
```

我无需知道 Kallie 代码的 JavaScript 内容，只需知道 dasher 是接口部分，是我所希望使用的变量和函数。我也无需知道 dasher 内部的具体实现方式，只需知道 dasher 函数的作用即可。当然，读者也可能确实对 dasher 内部的具体实现感兴趣，但是我认为，没有必要为了使
234 用该接口而去费力地理解其内部的具体实现。

以下为如何使用 dasher 函数来更新猜数字游戏。



代码清单 13-10　使用 Kallie 的代码
(http://jsbin.com/zusodu/edit?html,js,console)

```
var getGuesser = function (lowest, highest) {
    var secret = between(lowest, highest);
    return function (userNumber) {
        var msg;
        if (userNumber === secret) {
            msg = dasher("Well done!");   //使用 dasher 分隔消息里的各个字符
            return spacer.wrap(msg, msg.length + 4, "=");
        } else if (userNumber> secret) {
            msg = dasher("Too high!");
            return spacer.wrap(msg, msg.length+4,"+"); //使用 spacer.wrap
                                                       //将一个字符作为消息
                                                       //的开始和结束标志
        } else {
            msg = dasher("Too low!");
            return spacer.wrap(msg, msg.length + 4, "-");
        }
    };
};
var guess = getGuesser(5, 10);
```

上述代码编写得很满意！下面，我们运行程序来进行数字猜测。错误！什么？spacer.wrap 去哪里了？报错信息如图 13-12 所示。

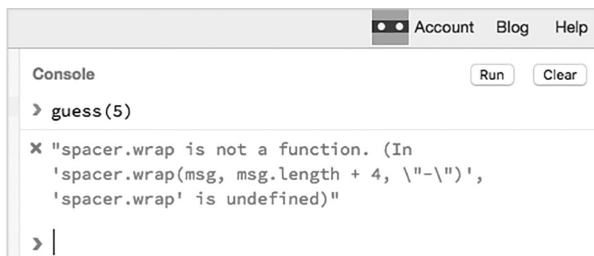


图 13-12　报错信息：程序找不到 spacer.wrap

spacer.wrap 似乎出问题了。令人不解的是，作为 spacer 名字空间的一部分，spacer.wrap 从第 7 章以来一直很好用，为什么现在出问题了？仔细检查 Kallie 的模块，从她的代码清单中，终于发现了问题所在。



代码清单 13-11　Kallie 的格式代码
(http://jsbin.com/soxeke/edit?js,console)

```
var spreader = function (text, character) {   //定义一个函数，用指定字符均
                                               //匀地将文本分散开
    return text.split("").join(character);                                235
};
var spacer = function (text) {    //定义一个用空格均匀分散文本的函数
    return spreader(text, " ");
};
var dasher = function (text) {    //定义一个用短划线均匀分散文本的函数
    return spreader(text, "-");
};
```

Kallie 的代码不仅包括 dasher 函数，还包括 spacer 函数。她使用了三个全局变量：spreader、spacer 和 dasher。我的 spacer 命名空间也使用了一个全局变量 spacer。现在，她的 spacer 覆盖了我的 spacer，如图 13-13 所示。

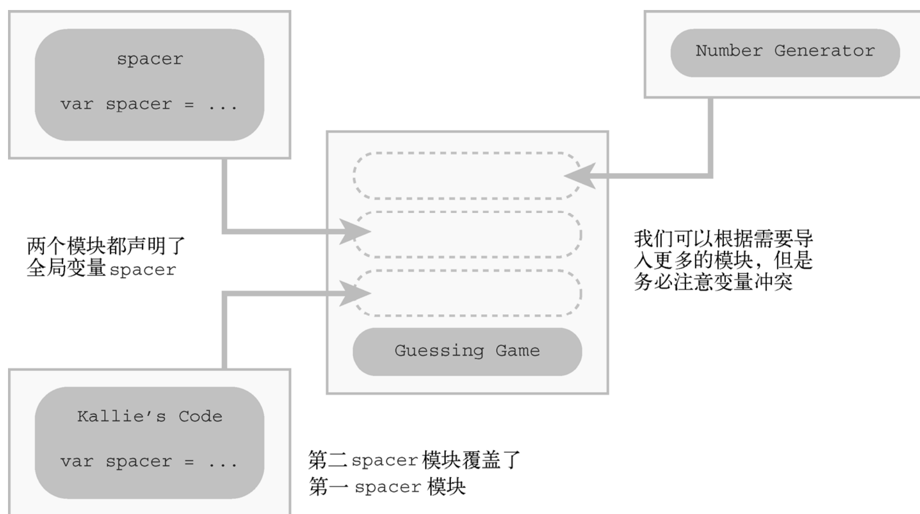


图 13-13　在两个模块中声明了的相同全局变量，从而导致冲突

问题并不在于函数的实现部分（函数本身运行良好），而在于如何让函数在模块中依然可用。编写各个模块的程序员都有可能会添加一些变量到全局作用域中，常常有可能在多个变量声明中使用相同的变量名称，从而造成冲突。

13.5.1　变量冲突

至少使用一个全局变量才能使模块在全局发挥作用，但如果同一变量被多个模块使用就会发生混战？好吧，混战的结果是最后一个模块赢了，它会把自己的值分配给该变量。在猜数字游戏代码中，加载了两个使用全局 spacer 变量的模块，并在程序中使用该变量。下面的 236
代码段显示了第二个 spacer 如何覆盖第一个 spacer，导致程序在尝试使用 spacer.wrap 时出现错误。

```
// 第一个模块的代码
var spacer = {
line : …,
wrap : …,
box : …
};
// 第二个模块的代码
var spacer = function (text) { … };
// 后面的代码
spacer.wrap(msg, msg.length + 4, "="); // 错误！找不到 spacer.wrap
```

如上例所见，当一个变量声明覆盖了另一个变量声明时，我们称之为变量冲突，结果是第二个变量声明彻底打败了第一个变量声明。现在，我们可以理解为什么声明很多全局变量就会污染全局命名空间。因为声明的变量越多，冲突的机会就越大。针对上述问题，本书在下一小节将讨论如何使用命名空间来减少全局变量的数量，在第 13.6 节中，将讨论立即调用函数表达式。

13.5.2 通过使用命名空间减少冲突

spacer 命名空间模块目前运行良好，因为该模块只使用了一个全局变量，而不是把全局变量分散到 line、wrap 和 box 函数。spacer 命名空间使用一个对象作为命名空间，并将函数分配给该对象的属性，然后将对象分配给单个变量 spacer。Kallie 为她的模块所造成的污染问题深感抱歉——她太忙了，疏忽了这个问题，现在 Kallie 用命名空间更新了这个模块，如以下代码清单 13-12 所示。



代码清单 13-12 Kallie 在命名空间中的格式化代码 (http://jsbin.com/moheka/edit?js,console)

```
var kalliesCode = {     //将函数封装在对象中，并将对象分配给单个全局变量
    spreader: function (text, character) {
        return text.split("").join(character);
    },
    spacer: function (text) {     //将 spacer 函数设置为命名空间对象的属性
        return kalliesCode.spreader(text, " ");
    },
    dasher: function (text) {
        return kalliesCode.spreader(text,"-"); //使用圆点运算符调用 kalliesCode
                                               //函数
    }
};
```

237

请更新猜数字游戏的 HTML，以便导入 Kallie 更新后的模块（代码清单 13-13），更新后的 JavaScript 代码从 kalliesCode 命名空间（代码清单 13-14）中调用 dasher 函数。



代码清单 13-13 导入 Kallie 的命名空间（HTML） (http://jsbin.com/seqahi/edit?html,js,console)

```
<!-- 数字生成器 -->
```

```
<script src="http://output.jsbin.com/qezoce.js"></script>
<!-- spacer -->
<script src="http://output.jsbin.com/juneqo.js"></script>
<!--Kallie's code -->
<script src="http://output.jsbin.com/moheka.js"></script>
```



代码清单 13-14　使用 Kallie 的代码 (http://jsbin.com/seqahi/edit?html,js,console)

```
var getGuesser = function (lowest, highest) {
    var secret = between(lowest, highest);
    return function (userNumber) {
        var msg;
        if (userNumber === secret) {
            msg = kalliesCode.dasher("Well done!");//从 kalliesCode 命名空
                                                   //间中调用 dasher 函数
            return spacer.wrap(msg, msg.length + 4, "=");
        } else if (userNumber> secret) {
            msg = kalliesCode.dasher("Too high!");
            return spacer.wrap(msg,msg.length+4,"+"); //顺利地使用 spacer.wrap
        } else {
            msg =kalliesCode.dasher("Too low!");
            return spacer.wrap(msg, msg.length + 4, "-");
        }
    };
};
var guess = getGuesser(5, 10);
```

请按照清单所示链接到 JS Bin，再次运行程序，就会看到以下显示。你会发现，通过使用命名空间，问题已经得到完美解决。

```
= W-e-l-l- -d-o-n-e-! =
```

使用函数是避免污染全局命名空间的另一种重要方式，将在下一节讲述。这种函数有一个好听的名字，就是……

13.6　立即调用函数表达式（IIFE）

立即调用函数表达式？这个标题看似有点儿匪夷所思，希望它能激起读者的好奇心，而不是迷惘和恐惧。立即调用函数表达式是指程序可以直接调用的函数，即不需使用变量也可以调用。这样做的意义何在？ 238

在第 13.5 节中，我导入了一个由老朋友 Kallie 编写的代码模块。但不幸的是，Kallie 程序中的全局变量与我的全局变量冲突，并且覆盖了我心爱的 spacer 命名空间。汲取这个严重教训，我下定决心要竭尽全力遏制全局命名空间的污染问题，必须检查代码，找到那些可以删除的全局变量。

首先，检查在本章前文中出现的测验应用程序。以下代码清单显示了代码的结构，包括全局变量、局部变量和接口对象。



代码清单 13-15　随机测验问题包括两个全局变量 (http://jsbin.com/ponogi/edit?html,js,console)

```
var getQuiz= function () {     //定义一个函数来创建局部作用域并将其分配给全局变
                               //量 getQuiz
    var qIndex = 0;       //在函数内使用 var 来声明局部变量
    var questions = [ … ];
    var getQuestion = function () { … };
    var checkAnswer = function (userAnswer) { … };
    return {              //返回一个接口对象，让用户访问某些函数
        quizMe: getQuestion,
        submit: checkAnswer
    };
};
var quiz= getQuiz();     //调用 getQuiz 并将返回的接口分配给全局变量 quiz
```

在以上程序中，声明了两个全局变量：getQuiz 和 quiz。在第一行代码中，定义一个函数并将其分配给 getQuiz，如下所示：

```
var getQuiz = function () { … };
```

在最后一行代码中，立即调用了该函数：

```
var quiz = getQuiz();
```

上述代码中所声明的 getQuiz 是一个全局变量，它就是污染源。在后续程序中，getQuiz 仅仅使用了一次，在其他时候它犹如一股恶臭的气味四处弥漫。别忘了，任何一个全局变量的存在都会给其他全局变量带来风险。这些全局变量可能在很小的代码段中运行良好，但是随着程序的增长和模块的创建和导入，这些全局变量就像蝴蝶扇动翅膀，引发
239 程序全盘崩溃。

友情提示：

（1）尽量避免声明全局变量，以免污染全局命名空间。

（2）可以通过使用对象作为命名空间的方法来减少全局变量的数量。

（3）尽量在函数内部声明局部变量。

使用立即调用函数表达式可以将全局变量的数量减半。要理解如何使用 IIFE 优化程序，请考虑以下内容：

- 识别函数表达式
- 调用函数
- 立即调用函数表达式
- 从 IIFE 返回信息

13.6.1　识别函数表达式

在本书中，从第 4 章就开始使用函数表达式。我们可以将函数表达式分配给变量和属性，函数表达式可以作为参数传递，还可以从其他函数返回函数表达式，如下所示：

```
var show = function (message) {            //
    console.log(message);                  // 分配给一个变量
};                                         //
var namespace = {
    show: function (message) {             //
        console.log(message);              // 分配给一个属性
    }                                      //
};
tweets.forEach(function (message) {        //
        console.log(message);              // 作为参数传递给 forEach
});                                        //
var getFunction = function () {
    var localMessage = "Hello Local!";
    return function () {                   //
        console.log(localMessage);         // 从一个函数返回
    };                                     //
};
```

我们已经学过使用函数来创建可以按需调用的代码块，也学过使用函数来创建局部作用域，因此使用函数可以隐藏某些变量，以免用户和其他程序员对其窥探和调整，下面介绍具体做法。

13.6.2　调用函数

调用函数需要使用函数调用运算符()，即一对圆括号。下面我们使用圆括号来调用上一节中的 4 个函数表达式。

```
show("Hello World!");                // 调用 show 函数
namespace.show("Hello World!");      // 调用 show 方法
// 被 forEach 自动调用
var show = getFunction();            // getFunction 返回一个函数
show();                              // 调用这个被返回的函数
```

240

将参数放在调用运算符——圆括号之间，就可以将该参数传递给函数。从函数返回的任何值都会替换该函数调用。

13.6.3　立即调用函数表达式

其实，不需要为了调用一个函数就专门将其分配给一个变量。只需将该函数表达式放在圆括号内，然后在该括号后面附加一个函数调用运算符，如下所示：

```
(function () {
    console.log("Hello World!");
```

```
})( );
```

图 13-14 是对以上代码的图示。

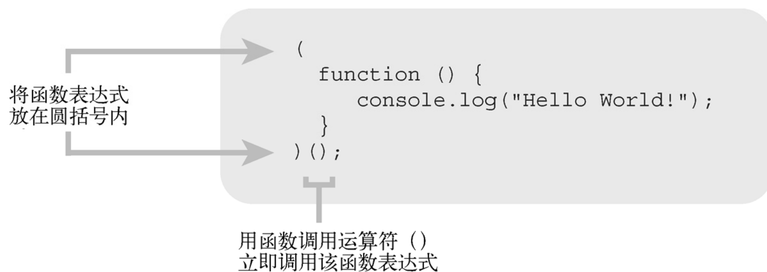


图 13-14　立即调用函数表达式

这个函数体内的代码立即运行，而且没有使用一个全局变量。可以说这个函数表达式对全局命名空间没有造成任何影响。让我们一起来呼吸这无污染的新鲜空气！

13.6.4　从 IIFE 返回信息

我们看到，立即调用函数不仅可以减少污染，而且还可以很好地隐藏私有变量。其实，立即调用函数还有更多作用。如同其他函数一样，立即调用函数还可以返回值，例如可以返回一个接口对象，通过接口对象我们可以对函数内部的秘密内容进行受控访问（请注意：必须仔细考虑函数内部的哪些内容可以对外公开）。

以下代码清单 13-16 展示了 IIFE 如何返回一个接口对象，并将其分配给一个变量。可以通过访问控制台上的 quiz 接口来运行该程序，如下所示：

```
>quiz.quizMe()
12 x 12
>quiz.submit("144")
Correct!
```

241



代码清单 13-16　在测验应用程序中使用 IIFE (http://jsbin.com/titano/edit?html,js,console)

```
var quiz = (function () {      //将函数放在圆括号内
    var qIndex = 0;            //隐藏该函数内的私有变量
    var questions = [ … ];
    var getQuestion = function () { … };
    var checkAnswer = function (userAnswer) { … };
    return {                   //返回一个公共的接口对象
        quizMe: getQuestion,
        submit: checkAnswer
    };
})();                          //在上述圆括号后面附加一个()来调用该函数
```

在以上代码中，先定义一个函数，并立即调用该函数，然后将返回的对象分配给 quiz 变量。从下面对比中可以看到，使用立即调用函数表达式可以减少使用不必要的变量

getQuiz。

我们并没有采用以下方式：

```
var getQuiz= function () {
    /* 私有化 */
    return interface;
};
var quiz = getQuiz();
```

而是采用了以下方式：

```
var quiz = (function () {
    /* 私有化 */
    return interface;
})();
```

在游戏 The Crypt 中，使用了许多构造函数、函数和对象。既然我们已经明白了使用模块和立即调用函数表达式的优势，那就让我们一起将游戏代码拆分成模块吧！

13.7　The Crypt——将代码分解成模块

正如本章开始所述，The Crypt 游戏的代码变得越来越长，因此非常有必要将程序分解成多个模块。通过学习本章内容，我们已经知道为什么要将代码分解成模块以及如何进行分解，现在是一展身手的时候了。

以下代码清单 13-17 显示了 5 个 HTML script 元素，用于加载不同的模块，从而逐步构建完整的游戏。 242



代码清单 13-17　为 The Crypt 导入模块（HTML）
(http://jsbin.com/zikuta/edit?html,js,console)

```
<!–spacer 模块 -->
<script src="http://output.jsbin.com/juneqo.js"></script>
<!–玩家构造器 -->
<script src="http://output.jsbin.com/nubijex.js"></script>
<!–场所构造器-->
<script src="http://output.jsbin.com/dofuci.js"></script>
<!–地图代码 -->
<script src="http://output.jsbin.com/dipaxo.js"></script>
<!–游戏初始化 -->
<script src="http://output.jsbin.com/fisupe.js"></script>
```

请读者注意，spacer 模块的地址就是其在本章中一直使用的地址，这就是模块的美妙之处！程序员可以在多个项目中使用同一个文件。如果在 spacer 命名空间中添加新的格式化函数，那么在已导入该模块的所有项目中，该函数会立即生效。

以下代码清单 13-18 显示了启动游戏所需要的单行 JavaScript 语句。



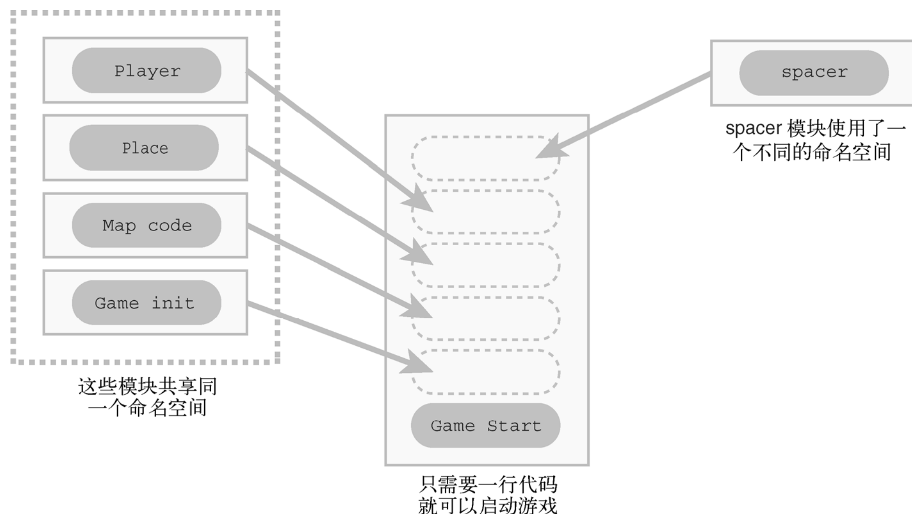
代码清单 13-18　为 The Crypt 导入模块
(http://jsbin.com/zikuta/edit?html,js,console)

```
var game = theCrypt.getGame();
```

以模块化形式开始游戏应该没有问题，因为所有功能都定义在模块中，仅仅一行代码就能够启动游戏。

除了 spacer，游戏 The Crypt 中其他模块都共享一个命名空间 theCrypt（图 13-15）。通过调用 theCrypt 命名空间的 getGame 方法启动游戏：

```
theCrypt.getGame();
```



243 图 13-15　游戏 The Crypt 中导入了 5 个模块，其中 4 个模块共享同一个命名空间

现在，我们知道使用命名空间能够有效减少全局变量，并且能够对相关函数进行分组。下面介绍如何让不同模块文件中的代码共享同一个命名空间。

13.7.1　跨模块共享命名空间

使用本章中学到的内容，程序员想要实现以下操作：

- 构成 The Crypt 的所有模块都能够使用同一个全局命名空间 theCrypt。
- 仅向该命名空间分配其他模块所需的属性和函数。
- 将所有其他内容隐藏在函数的局部作用域中。

正如给对象添加属性，这些模块给命名空间 theCrypt 添加属性，如下所示：

```
theCrypt.Player = Player;
```

既然众多模块要共享同一个命名空间，那么首先要确定由哪个模块来创建这个命名空间。如果程序访问到未声明的变量就会导致错误。为了避免这种错误，程序员特意把声明 theCrypt 的模块排在第一位，但是这种为加载模块排序的做法显然很不方便。好在我们拥有一个特殊的全局对象——window。在浏览器中，所有全局变量会自动分配给 window。

1．全局对象：window

window 对象是 JavaScript 在浏览器中使用全局变量的一种方式。读者可以在控制台提示符下测试 window 对象并检查某变量是否已经声明为全局变量，并测试 window 对象：

```
>var test = "Hi"
undefined
>test
"Hi"
>window.test
"Hi"
```

在以上交互中，声明了一个全局变量 test，并自动分配为 window 对象的属性。再次回到控制台提示符下，尝试访问一个不存在的全局变量：

```
>theCrypt
"Can't find variable: theCrypt"
```

系统马上抛出了一条出错信息（尽管你的浏览器显示的错误消息可能会略有不同）。但是，如果访问 window 一个尚不存在的属性，却不会抛出错误信息，而是返回 undefined：

```
>window.theCrypt
undefined
```

由此可见，我们可以使用 window 对象来检查 theCrypt 是否已经被声明为命名空间，这样就不会引起任何错误。 244

2．使用 window 对象来检查全局变量

下一个代码清单是 Player 模块，将 Player 构造函数及其外围代码打包为一个整体来实现其模块化。



代码清单 13-19　将 Player 构造函数构建为模块 (http://jsbin.com/nubijex/edit?js)

```
(function () {          //将 Player 构造函数包装在一个函数中以创建局部作用域
   var Player = function (name, health){  //不允许用户接触到 Player 构造函数
      /*程序实现无变化*/
   };
   if(window.theCrypt === undefined){//确保有一个名为 theCrypt 的全局命名空间
      window.theCrypt = {};
   }
   theCrypt.Player = Player;          //将 Player 构造函数分配给该命名空间
})();          //立即执行函数表达式
```

在以上代码清单 13-19 中，Player 模块通过检查 window 对象来查看 theCrypt 是否已声明为全局变量。如果 Player 模块找不到 theCrypt，那么就对其进行声明：

```
if (window.theCrypt === undefined) {
   window.theCrypt = {};
```

```
}
```

一旦该模块确定命名空间已存在，就将 Player 构造函数添加为命名空间的一个属性：

```
theCrypt.Player = Player;
```

我们看到，借助全局变量 theCrypt，Player 构造函数已经可以供其他代码使用。

以下代码清单 13-20 显示 Place 模块，同样使用了 theCrypt 全局命名空间。



代码清单 13-20　将 Place 构造函数构建为模块 (http://jsbin.com/dofuci/edit?js)

```
(function () {
   var Place = function (title, description) {
      /*程序实现无变化*/
   };
   if (window.theCrypt === undefined) {
      window.theCrypt = {};
   }
      theCrypt.Place = Place;      //将 Place 构造函数分配给该命名空间
})();
```

245

让每个模块都检查 theCrypt 是否为 window 对象的全局变量，这意味着并不因为某个模块能够创建命名空间，就必须将其放在第一位进行加载。所有模块都要检查 theCrypt，如果第一个加载模块找不到 theCrypt 命名空间，该模块会在使用之前先创建 theCrypt。之后加载的模块在找到 theCrypt 命名空间后，将模块作为属性分配给该命名空间。

3．将命名空间属性分配给局部变量

命名空间能够有效减少全局变量，但是使用命名空间的属性往往意味着要输入一个很长的名字，这点让使用者感到有些麻烦。例如，地图代码模块需要在 The Crypt 中创建场所，使用 5 个字母的 Place 构造函数即可：

```
var kitchen = new Place("The Kitchen", "You are in a kitchen…");
```

但是现在将 Place 构造函数移动到 theCrypt 命名空间中，对于在地图代码中更改对 Place 的所有引用确实方便了，然而却需要输入 13 个字母，theCrypt.Place，如下所示：

```
var kitchen = new theCrypt.Place("The Kitchen", "You are in a kitchen…");
var library = new theCrypt.Place("The Old Library", "You are in a lib…");
```

事实上，可以通过创建一个局部变量 Place 来解决上述问题，如下所示：

```
var Place = theCrypt.Place;
var kitchen = new Place("The Kitchen", "You are in a kitchen…");
```

如果程序员需要多次使用命名空间的某个属性，这是一种可以减少打字的好方法。

在本章中，我们已经学会如何将代码段移动到自己的模块中，下一步可以仔细考虑每个模块的功能。当任务变得更加细致和具体时，这些模块还可以分成更小的模块吗？当程序中

常常出现变化的数据时，我们该使用哪些代码模式来处理和显示这些不断变化的数据呢？请读者在后续三章内容中寻找答案：模型、视图和控制器。

13.8　本章小结

- 将程序分割成模块，成为可以独立加载的代码段。
- 在多个项目中使用相同的模块。
- 在 JS Bin 中使用 HTML 脚本元素加载模块：

```
<script src="path/to/module.js"></script>
```

- 在 script 元素的 src 属性中指定模块文件的位置。
- 变量冲突就是程序中的一个变量被另一个变量覆盖，使用命名空间和立即执行函数表达式可以减少变量冲突。 246
- 如何创建立即执行函数表达式：在圆括号中包含函数定义并附加函数调用运算符()。

```
(function () {
    // 立即执行的代码
})();
```

- 从立即执行函数表达式返回一个接口，这种方式称为模块模式：

```
var game = (function () {
    //以下是局部变量
    return {
        //接口方法
    };
})();
```

- 确保已经加载了其他模块所依赖的全部模块。 247

第 14 章　模型：使用数据

本章内容包括:

- 从数据构建模型
- 在多个项目中使用相同的数据
- 在数据文件之间切换
- 在游戏 The Crypt 中指定地图数据

在第 13 章中，我们学习了如何使用模块将程序分解成单独的文件。之后，就可以独立操作模块，在模块之间轻松切换，并在多个项目中反复使用这些模块，甚至可以发布自己编写的模块，并导入他人已发布的模块。

本章继续坚持模块化和代码复用的思想。在本章中，读者将学习如何从构造函数和函数中移出数据。首先，读者将学习如何使用简单的方式呈现数据，以便该数据可供多个应用程序使用，不论这些应用程序是采用何种语言编写。随后，考虑如何将该数据提供给构造函数和函数，以构建能够添加额外功能的模型。最后，练习为 The Crypt 定义地图数据，为玩家增加挑战，使游戏更具吸引力。

14.1　构建健身应用程序——数据和模型

作为健身应用程序开发团队的一员，需要考虑以下客户需求：关注健康的用户需要跟踪他们的锻炼情况，把每次锻炼的日期和持续时间都记录下来，如下所示。

```
Mahesha
120 minutes on February 5th, 2017
35 minutes on February 6th, 2017
2 hours 35 minutes so far.
Great work!
```

程序开发团队正在使用 Python 编程语言编写 Android 版本的应用程序，同时使用 Swift 编写 iOS 版本的应用程序，还在使用 JavaScript 编写网页版本的应用程序，以上三个版本的应用程序将使用相同的数据，如图 14-1 所示。

构建以上应用程序需要完成以下任务：

（1）以字符串形式检索用户数据。

（2）将用户数据转换为用户模型。

（3）显示用户数据。

（4）为用户提供添加会话的接口。

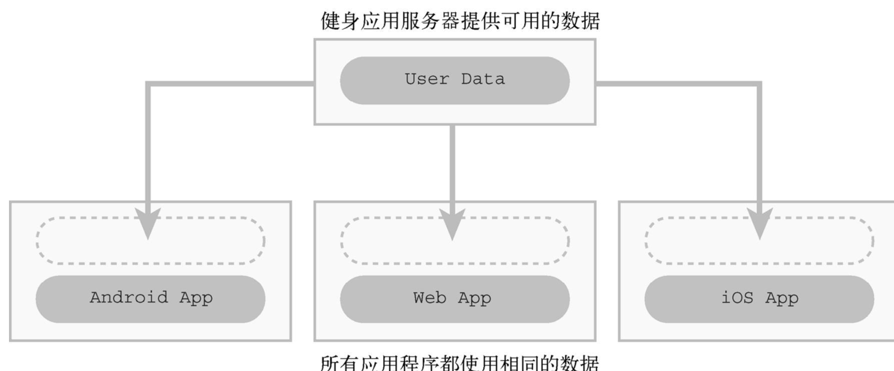


图 14-1　不同版本的应用程序都使用相同的数据

将数据作为文本在互联网上传输。文本的格式是 JSON，有关 JSON 的内容将在第 20 章详述。我们将会看到，这种文本格式很容易转换为 JavaScript 对象。作为开发团队的一员，我们现在专注于以上第二项任务：将用户数据转换为用户模型。 249

14.1.1　定义 User 构造函数

作为开发团队的一员，我们的任务是编写 JavaScript 代码，为健身应用的用户建模，该模型需要执行以下操作：

（1）存储用户名。

（2）存储用户每一次锻炼的列表，每次锻炼都包含日期和持续时间。

（3）包含将每次锻炼添加到列表的方法。

（4）包含用于检索相关用户数据的方法。

代码清单 14-1 显示初始的构造函数。在控制台对它进行调试，交互如下：

```
> var user = new User("Mahesha")
undefined
> user.addSession("2017-02-05", 120)
120
> user.addSession("2017-02-06", 35)
155
> user.getData().total
155
```



代码清单 14-1　User 构造函数
(http://jsbin.com/suzala/edit?js,console)

```
var User = function (name) {
    var sessions = [];          //声明私有变量
    var totalDuration = 0;
    this.addSession = function (sessionDate, duration) {
        sessions.push({         //将 session 对象添加到 sessions 数组
            "sessionDate" : sessionDate,
            "duration" : duration
```

```
            });
            totalDuration += duration;  //将当前一次锻炼的持续时间加到总计时间
            return totalDuration;
        };
        this.getData = function () {    //定义一个函数，用于检索用户数据
            return {
                "name" : name,
                "total" : totalDuration,
                "sessions": sessions.slice()    //使用 slice 复制 sessions 数组
            };
        };
    }
```

以上 User 构造函数中，包括一个 name 形参、几个使用 var 声明的私有变量、以及两个赋值给 this 对象的公共方法 addSession 和 getData。getData 方法使用没有参数的 slice 来抓取
250 sessions 数组的副本。有关数组方法（如 slice）的详细讲解，请读者参阅第 8 章内容。提供 session 信息的副本可以防止用户在 addSession 方法以外的地方对 sessions 数组的内容进行调整。getData 返回的对象还包括一个 total 属性，是多次锻炼的总持续时间。

当使用 User 构造函数创建 JavaScript 对象时，就创建了一个用户模型。模型不仅仅是数据，还包括用于管理数据的私有变量和公共方法，如图 14-2 所示。

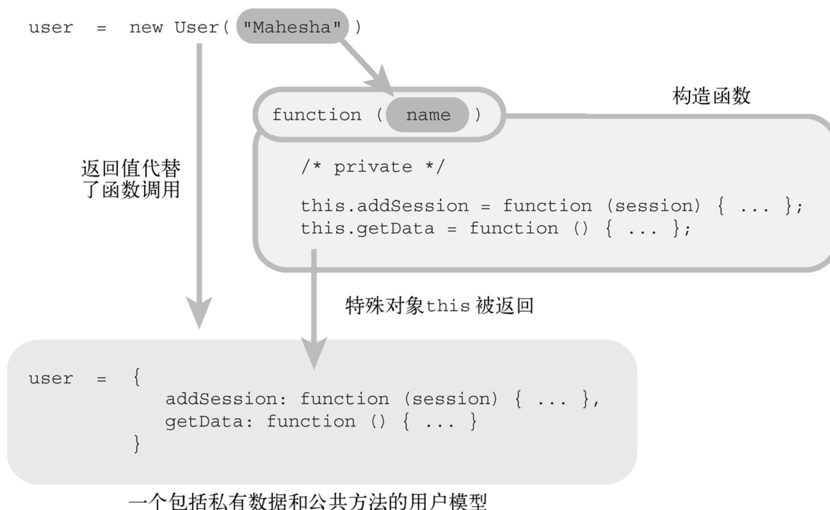


图 14-2　由 User 构造函数创建的用户模型

既然模型不仅仅是数据，那么数据又是什么？

14.1.2　体会把数据作为 JavaScript 对象的感觉

以下用户数据就是一个简单的 JavaScript 对象：

```
var userData = {
    "name" : "Mahesha",
```

```
        "sessions" : [
            { "sessionDate" : "2017-02-05", "duration" : 120 },
            { "sessionDate" : "2017-02-06", "duration" : 35 },
            { "sessionDate" : "2017-02-06", "duration" : 45 }
        ]
    };
```

我们可以访问该对象的属性，例如 userData.name，但是它依然只是数据，并不具有模型的其他功能，例如由 User 构造函数创建的用户模型中的 addSession 或 getData 方法。在图 14-3 中再次展示了数据对象，以便与图 14-2 进行比较。 251

```
user  =  {
             name: "Mahesha",
             sessions: [ ... ]
         }
```

用户数据作为简单的JavaScript 对象

图 14-3　用户数据作为简单的 JavaScript 对象

正如在其他编程语言中一样，JavaScript 常常将简单的对象作为数据格式，在网络上尤其如此。但是，为了充分利用用户模型 addSession 和 getData 提供的额外方法，我们需要定义一个函数，把基本数据对象构建成为用户模型。

14.1.3　将数据转换为用户模型

在以下代码清单 14-3 中，定义了将单个用户的数据作为 JavaScript 对象的 buildUser 函数，并通过调用 User 构造函数来创建模型。通过从 JavaScript 对象中创建用户模型来测试 buildUser 函数，为创建的用户添加一次额外的锻炼就会在控制台上输出以下信息：

```
> 240
```

我们正在使用代码清单 14-1 中的 User 构造函数，因此可以通过向项目中添加 script 元素的方法（参见第 13 章）导入该构造函数，如代码清单 14-2 所示。



代码清单 14-2　从用户数据构建用户模型的函数（HTML）(http://jsbin.com/zenire/edit?html,js,console)

```
<!—用户构造器 -->
<script src="http://output.jsbin.com/suzala.js"></script>
```



代码清单 14-3　从用户数据构建用户模型的函数（http://jsbin.com/zenire/edit?html,js,console）

```
var buildUser = function (userData) {
    var user = new User(userData.name);  //使用 User 构造函数创建新的用户对象
    userData.sessions.forEach(function (session) {
        user.addSession(
        session.sessionDate, session.duration);      //添加每一次锻炼
```

252

```
    });
    return user;          //返回新创建的用户模型
};
var userData = {          //创建 JavaScript 对象以保存用户数据
    "name" : "Mahesha",
    "sessions" : [
        {"sessionDate": "2017-02-05", "duration": 120},
        {"sessionDate": "2017-02-06", "duration": 35},
        {"sessionDate": "2017-02-06", "duration": 45}
    ]
};
var user = buildUser(userData);        //调用 buildUser 从数据创建模型
console.log(user.addSession("2017-02-15",40));//添加一次锻炼并记录返回的总时间
```

在以上代码中，通过使用 buildUser 函数，将普通用户数据升级为增强的用户模型，普通用户数据是一个简单的 JavaScript 对象，而增强的用户模型不但将数据隐藏为私有变量，而且添加了管理模型状态和访问数据副本的方法。

14.1.4 下一步目标

以上应用程序开发工作已经满足本章的要求，现在将该应用程序的完整要求重申一下：

（1）以字符串形式检索用户数据。

（2）将用户数据转换为用户模型。

（3）显示用户数据。

（4）为用户提供添加会话的界面。

我们已完成上述第二项要求，其他几项要求将会在本书后续内容中得以落实。现在，再来看看游戏 The Crypt，我们是否可以像在健身应用中那样从用户模型中分离出用户数据，即从游戏 The Crypt 场所模型中分离出地图数据？

14.2 The Crypt——从游戏中分离出地图数据

在本节中，我们将把从健身应用程序中学到的知识应用到游戏 The Crypt 中，主要完成以下任务：

（1）使用基本的 JavaScript 对象来表示游戏中的地图数据。

（2）向地图数据中的出口添加挑战内容。

（3）使用方法更新 Place 构造函数以设置和获取挑战内容。

（4）编写一个函数，用来从地图数据构建场所模型。

目前，我们已经学会在程序中采用以下两个步骤手动构建 The Crypt 地图：在我们创建的每个地方调用 Place 构造函数，然后通过调用方法来添加物品和出口。

```
// 创建两个地方
var kitchen = new Place(
    "The Kitchen",
```

```
    "You are in a kitchen. There is a disturbing smell."
);
var library = new Place(
    "The Old Library",
    "You are in a library. Dusty books line the walls."
);
kitchen.addItem("a piece of cheese");   // 单独添加物品
library.addItem("a rusty key"); //
kitchen.addExit("south", library);      //单独添加出口
library.addExit("north", kitchen);      //
```
253

在采用以上方法创建的地图中，地图数据（地点描述、出口和物品）与创建游戏对象所使用的 JavaScript 捆绑在一起。在代码中唯一可以找到地图位置的地方就是调用 Place 构造函数。

```
var library = new Place(
    "The Old Library",                // 内部数据
    "You are in a library. Dusty books line the walls."  // 构造函数
);
```

但是，关于这些地点的出口和物品详细信息与地点本身是相互分离的。

在 14.1 节中我们看到，当数据以通用格式表示时，程序可以更容易地共享这些数据，因为其他程序和编程语言可以读取已经格式化为简单 JavaScript 对象的数据，但是并不能读取 Place 对象。如果能将原始地图数据与构造函数分开，与方法（例如 addItem 和 addExit）等分开，就会使定义新地图、存储、切换和共享这些地图变得容易很多（图 14-4）。

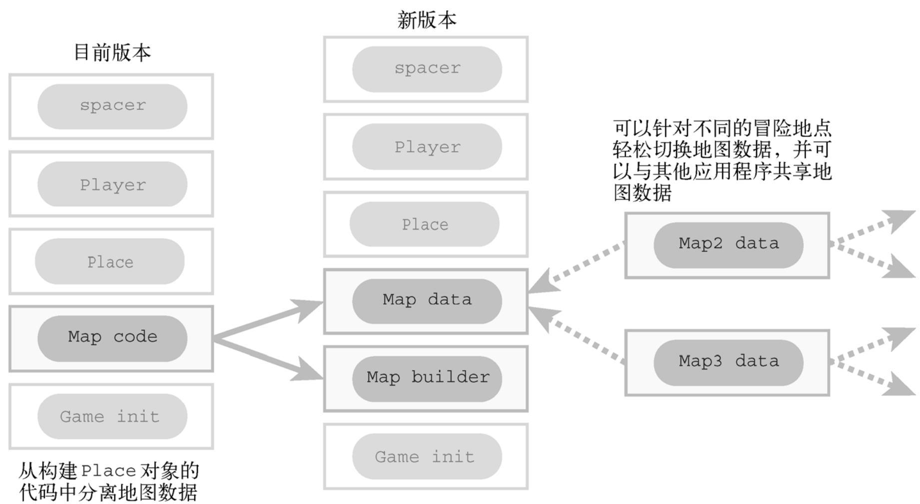


图 14-4　从地图构建器中拆分出地图数据可以使切换地图变得更加容易　254

为了实现以上数据和游戏代码的分离，我们必须首先决定数据将采取何种形式，然后编写一个函数将数据转换为游戏使用的 Place 模型。

14.2.1 地图数据

每一个 JavaScript 对象将代表一个地图，该对象包含标题、地点列表和开始位置名称。

```
{
    "title": "The Dark House",
    "firstPlace" : "The Kitchen",
    "places" : [
        //  place 对象的数组
    ]
};
```

places 数组中的每个地方也将是一个对象。以下代码段显示了一个这样的地方：

```
{
    "title" : "The Kitchen",
    "description" : "You are in a kitchen. There is a disturbing smell.",
    "items" : [ "a piece of cheese" ],
    "exits" : [
        { "direction" : "south", "to" : "The Old Library" },
        { "direction" : "west", "to" : "The Kitchen Garden" },
        { "direction" : "east", "to" : "The Kitchen Cupboard" }
    ]
};
```

我们可以看到，每个出口是一个对象，属性为出口方向以及该出口所指向地方的标题。这样的数据是紧凑的、可读的，并且可以保持物品和出口与它们所属的地方在一起。

以下列表显示了地图数据的一部分，在 JS Bin 上可见 4 个位置的完整代码。



代码清单 14-4　地图数据 (http://jsbin.com/qonoje/edit?js,console)

```
var mapData = {
    "title" : "The Dark House",      //给每张地图一个标题
    "firstPlace" : "The Kitchen",    //在地图中包含第一个位置的标题
    "places" : [                     //列出 places 数组中的所有 place 对象
        {
        "title" : "The Kitchen",
        "description":"You are in a kitchen.There is a disturbing smell.",
        "items" : [ "a piece of cheese" ],
        "exits" : [
            { "direction": "south", "to": "The Old Library" },
            { "direction": "west", "to": "The Kitchen Garden" },
255         { "direction": "east", "to": "The Kitchen Cupboard" }
            ]
        },
        {
            "title" : "The Old Library",
```

```
                "description":"You are in a library.Dusty books line the walls.",
                "items" : [ "a rusty key" ],
                "exits" : [
                    { "direction" : "north", "to" : "The Kitchen" }
                ]
            },
            {
                "title" : "The Kitchen Garden", /* JS Bin上有完整代码 */
            },
            {
                "title" : "The Kitchen Cupboard", /* JS Bin上有完整代码 */
            }
        ]
    };
```

以上地图数据只是对每个地方的描述，构建 Place 模型将会为其增添程序所期望的功能。

14.2.2　向 The Crypt 地图数据添加挑战

在游戏 The Crypt 中，玩家可以探索新奇的世界，可以在异国他乡发现物品，而且可以拾起这些物品，并据为己有。听上去很不错，但是似乎缺了点儿什么。这样的冒险更像是度假，过于轻松自在了，缺少了点儿刺激。冒险，就需要接受挑战！

为了使游戏更有趣，我们在出口处增加了挑战。即当玩家试图去某个方向时，可能会遇到一个问题需要解决。游戏交互如下所示：

```
> game.go("south")
*** A zombie sinks its teeth into your neck. ***
> game.use("a piece of cheese south")
*** The zombie is strangely resilient. ***
> game.use("holy water south")
*** The zombie disintegrates into a puddle of putrid goo. ***
> game.go("south")
The Old Library
You are in The Old Library…
```

这个挑战阻止玩家向南继续前进。为了迎接该挑战，玩家必须在此方向上使用一个特定的物品：

```
game.use("holy water south")
```
256

如果玩家手中没有上述必要的物品，就必须去其他方向冒险，应对其他挑战，直到玩家找到所需要的物品。那么，我们应该如何在游戏中创建挑战？

所有关于 The Crypt 中的冒险信息都需要呈现在地图数据中。目前，出口信息很简单，如下所示：

```
{ "direction" : "south", "to" : "The Old Library" }
```

以上出口信息中，包含一个方向和一个目的地标题。要给某个特定出口添加挑战，就要添加一个 challenge 属性，如下所示：

```
{
    "direction" : "south",
    "to" : "The Old Library",
        "challenge" : {
            "message" : "A zombie sinks its teeth into your neck.",
            "success" : "The zombie disintegrates into a puddle of goo.",
            "failure" : "The zombie is strangely resilient.",
            "requires" : "holy water",
            "itemConsumed" : true,
            "damage" : 20
        }
}
```

表 14-1 列出 challenge 对象的各个属性及其含义，以及该属性是否必需。

表 14-1　challenge 的各个属性

属性	含义	是否必需
message	当玩家试图朝该出口方向前进时显示给玩家的消息，告诉玩家需要应对的挑战	是
success	当玩家使用了需要应对挑战的物品时，向玩家显示的消息	是
failure	当玩家使用错误的物品来应对挑战时，向玩家显示的消息	是
requires	应对挑战所需的物品	是
itemConsumed	物品被使用后是否从玩家的物品列表中将其删除	否
damage	在玩家应对挑战之前就试图从出口方向前行时，从玩家的健康中减去的量	否

257 为了应对以上这些挑战，我们需要更新 Place 构造函数。

14.2.3　更新 Place 构造函数来应对挑战

我们必须更新 **Place** 模型来存放这些挑战。首先，需要一个对象来存储这些挑战，另外还需要方法 addChallenge 和 getChallenge，为指定方向添加和检索挑战。代码清单 14-5 显示了 Place 构造函数的更新之处。



代码清单 14-5　带有挑战的 Place 构造函数 (http://jsbin.com/ruviso/edit?js,console)

```
var Place = function (title, description) {
    // 其他变量
    var challenges = {};        //创建一个空对象来存储任何挑战
    // 其他方法
    this.addChallenge = function (direction, challenge) {
                                // addChallenge 方法为指定方向存放一个挑战
        challenges[direction] = challenge;
    };
```

```
        this.getChallenge=function (direction){     //getChallenge 方法返回指定
                                                    //方向所存储的挑战
            return challenges[direction];
        };
    };
```

在以上代码中，首先创建一个私有对象 challenges 来存储任何挑战。如同出口（exits）一样，我们使用方向（direction）作为存储挑战（challenges）的键（keys）。例如，玩家在向南移动之前必须应对挑战，挑战的细节将存储在 challenges["south"]中。

为了存储挑战，需要使用 addChallenge 方法；为了在指定方向检索挑战，需要使用 getChallenge 方法。

14.2.4　使用地图数据构建游戏地图

The Crypt 的实现使用了 Place 构造函数创建的模型，该模型与各出口相连。既然地图数据不再与游戏逻辑绑定，那么我们需要一种方法将地图转换为一组场所模型。

我们编写名为 buildMap 的函数，该函数将地图数据对象作为参数，创建地点模型，该模型与各出口相连。该函数返回地图上的第一个地点模型就是游戏的起点：

```
    var firstPlace = buildMap(mapData);
```

图 14-5 显示 buildMap 函数如何两次使用 forEach：首先创建地点模型，然后通过添加出口来连接各模型。 258

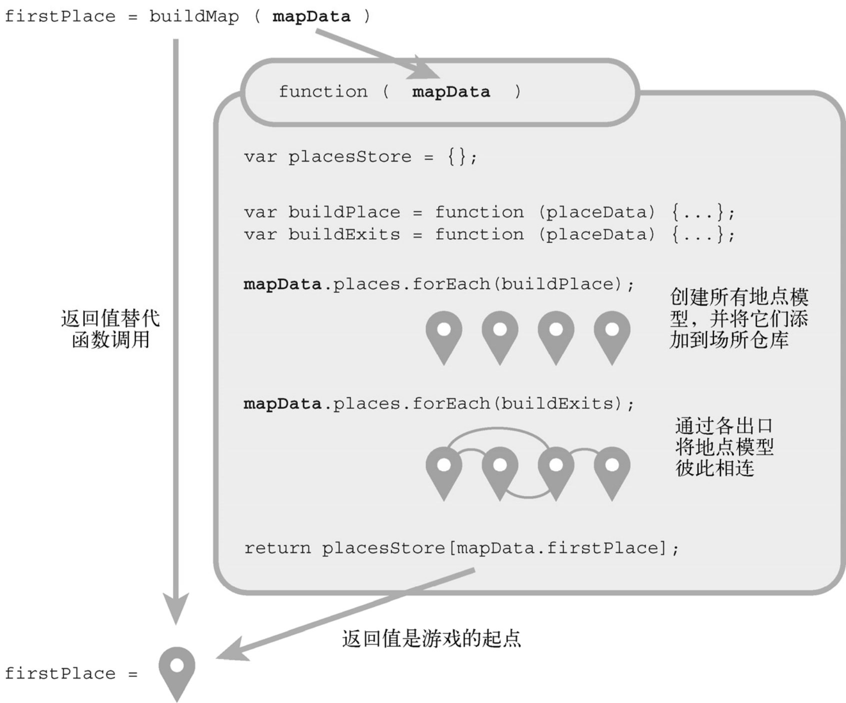


图 14-5　首先，创建所有的地点模型，然后通过其出口相互连接

buildMap 函数的代码如代码清单 14-6 所示，运行步骤如下所示：

（1）为每个地点创建模型（buildPlace）。

a 用标题和描述调用 Place 构造函数

b 将所有物品添加到新创建的地点模型

c 将地点模型放入地点仓库中

（2）为每个地点添加出口和挑战（buildExits）。

a 从地点仓库中检索地点模型

b 为地点数据中的每个模型添加出口

c 为地点数据中的每个出口添加挑战

（3）返回模型是游戏的起点。



代码清单 14-6　地图构建器
(http://jsbin.com/paqihi/edit?js,console)

259

```
var buildMap = function (mapData) {
    var placesStore = {};
    var buildPlace = function (placeData) {
        var place = new theCrypt.Place(//步骤 1a：用标题和描述调用 Place 构造函数
            placeData.title,
            placeData.description
        );
        if (placeData.items !== undefined) {    //步骤 1b：将所有物品添加到新
                                                //创建的地点模型
            placeData.items.forEach(place.addItem);
        }
        placesStore[placeData.title] = place;   //步骤 2a：从地点仓库中检索地
                                                //点模型
    };
    var buildExits = function (placeData) {
        var here = placesStore[placeData.title];//步骤 2b 为地点数据中的每个模
                                                //型添加出口
        if (placeData.exits !== undefined) {
            placeData.exits.forEach(function (exit) {
                var there = placesStore[exit.to];
                here.addExit(exit.direction, there); //步骤 2c：为地点数据中
                                                     //的每个出口添加一个挑战
                here.addChallenge(exit.direction, exit.challenge);
            });
        }
    };
    mapData.places.forEach(buildPlace);  //开始步骤 1：为每个地点创建一个模型
    mapData.places.forEach(buildExits);  //开始步骤 2：为每个地点添加出口
    return placesStore[mapData.firstPlace];     //步骤 3：返回游戏中第一个位置
                                                //的模型
};
```

在以上代码中，buildPlace 函数通过使用 Place 构造函数将单个地方的数据转换为地点模型。请注意，游戏模块中使用了全局命名空间 theCrypt（参见第 13 章），所以通过 theCrypt.Place 可以访问构造函数。首先要保证所有的地点模型都已经创建完毕，然后才可以通过出口让它们彼此相连。通过使用 forEach 迭代 mapData.places 数组，可以为地图数据中的每个地点调用 buildPlace，如下所示：

```
mapData.places.forEach(buildPlace);
```

在 buildPlace 中，将创建的每个地点都添加到 placesStore 对象。buildExits 函数将每个地点的数据分配给参数 placeData，并从 placesStore 中获取与之匹配的地点模型，如下所示：

```
var here = placesStore[placeData.title];
```

260

以上语句中，将该模型分配给 here 变量。因为地点模型中有一个 addExit 方法，所以可以使用 here.addExit 为当前模型添加出口。出口数据如下所示：

```
"exits" : [
   {    "direction": "south",
      "to": "The Old Library",
      "challenge" : {
         "message" : "A zombie sinks its teeth into your neck.",
         "success" : "The zombie disintegrates into a puddle of goo.",
         "failure" : "The zombie is strangely resilient.",
         "requires" : "holy water",
         "itemConsumed" : true,
         "damage" : 20
      }
   },
   { "direction": "west", "to": "The Kitchen Garden" },
   { "direction": "east", "to": "The Kitchen Cupboard" }
]
```

buildExits 通过出口数据得以运行，并为其找到的每个出口调用 addExit 和 addChallenge，如下所示：

```
placeData.exits.forEach(function (exit) {
   var there = placesStore[exit.to];
   here.addExit(exit.direction, there);
   here.addChallenge(exit.direction, exit.challenge);
});
```

在以上代码中，每个出口的 to 属性给出了该出口所指向目的地的标题。因此，通过使用 exit.to 作为键，可以在 placesStore 中找到该出口所指向目的地的模型。

无论地图数据中是否存在挑战，我们都可以向出口添加挑战，如下所示：

```
here.addChallenge(exit.direction, exit.challenge);
```

如果在地图数据中某个出口并没有挑战，那么 exit.challenge 的值为 undefined。在第 16

章，我们将编写挑战的代码。请注意，在使用挑战之前，应该先检查该挑战是否未定义。

通过为每个地点调用 buildPlaces 和 buildExits，我们已经创建了一个相互联系的地点模型地图，供勇敢的冒险家去探索。他们能如期找到财富与梦想，发现人生的启迪吗？还是会遭遇厄运？无限可能在向他们招手。无论如何，他们需要从某个地方开始启程，buildMap 函数的返回值就是地图上第一个地点模型，代码如下：

261

```
return placesStore[mapData.firstPlace];
```

我们已成功地将地图数据与游戏实施分离。这些数据现在是一种可以跨项目和跨编程语言，可以重复使用的数据形式。

如果单击代码清单 14-6 的 JS Bin 链接，就会看到完整的代码，该程序将 buildMap 函数包装在一个可以立即调用的函数表达式中，并将 buildMap 赋值给全局命名空间 theCrypt。因为这部分与其他所有模块的运行机制相同，所以并没有显示在本书的代码清单中。

14.2.5 运行游戏

我们需要对游戏初始化模块进行一个微小的调整，即对 buildMap 的调用进行更新，具体做法是从新的地图数据模块为 buildMap 函数传递数据。



代码清单 14-7 使用新的地图构建器 (http://jsbin.com/mogano/edit?js,console)

```
var getGame = function () {
   var render = function () { … };
   var firstPlace = theCrypt.buildMap(theCrypt.mapData);   //将地图数据
                                                          //传递给 buildMap 函数
   var player = new theCrypt.Player("Kandra", 50);
   player.addItem("The Sword of Doom");
   player.setPlace(firstPlace);
   render();
   //返回公共接口
   return {
      go: function (direction) { … },
      get: function () { … }
   };
};
```

buildMap 函数的返回值为冒险的起点，并将其设置为玩家的初始位置。

另外，还需要添加一个 HTML script 元素，用来在该游戏中导入地图数据模块。以下代码清单 14-8 显示了本游戏的最新版在 HTML 面板上的内容。



代码清单 14-8 使用地图构建器（HTML） (http://jsbin.com/rulayu/edit?html,console)

```
<!-- spacer -->
<script src="http://output.jsbin.com/juneqo.js"></script>
```

```
<!--玩家构建器 -->
<script src="http://output.jsbin.com/nubijex.js"></script>
<!--地点构建器-->
<script src="http://output.jsbin.com/ruviso.js"></script>
<!--地图数据(以带有挑战的厨房为例) -->
<script src="http://output.jsbin.com/jayici.js"></script>
<!--地图构建器 -->
<script src="http://output.jsbin.com/paqihi.js"></script>
<!--游戏初始化 -->
<script src="http://output.jsbin.com/mogano.js"></script>
```

262

试着运行一下吧！游戏操作应该还像以前一样，game.go 和 game.get 这两个命令依然可用。不必担心遭遇僵尸（其实僵尸还在，只不过它们潜伏在地图数据的阴影里，让你根本不会察觉）。在第 16 章，我们将学习把挑战也并入到游戏当中。

地图数据与地图构建代码分离的做法可以使数据在地图之间更容易切换。正因为我们已经做好了这样的分离，所以以下这款游戏就可以保持游戏代码不变，而只需要更新一个地图文件。（这是一个古老的绝地求生地图戏法！）

THE SPARROW

地图数据：http://jsbin.com/woniqo/edit?js

游戏：http://jsbin.com/dequzi/edit?console

在下一章内容中，我们将进一步贯彻模块化思路，将玩家和地点的显示与其模型分离开。

14.3　本章小结

- 以一种易于在各项目、应用和编程语言之间重复使用的形式来表示数据。简单的 JavaScript 对象是在网络上交换数据的常用格式。
- 将数据与程序逻辑分离，以便程序能够更容易地切换数据源。
- 定义一些模型，通过添加功能和隐藏私有变量来增强数据的可用性，为程序的其他部分使用该模型做好准备。
- 定义一些函数，从数据创建模型。 263

第 15 章　视图：显示数据

本章内容包括：

■ 使用视图呈现数据
■ 从构造函数中删除显示信息的代码
■ 将同一模型传递到多个视图

控制台并不是唯一可以为用户显示信息的地方。我们希望看到程序能够在网页上输出数据，或者在电脑桌面、手机应用程序、电子邮件和打印文件中输出数据。即使信息显示在控制台上，也希望输出格式能够灵活多样，我们不仅想看到一个简单的文本版本，还想看到一个带有方框和边框的花式版本。可是，我们又不想仅仅为了改变输出形式就重新编写大部分的程序。

视图是专注于显示信息的代码模块，这些模块获取数据，并基于数据创建可视化输出。视图可能会包含按钮和文本框等控件，相关内容将留至本书第三部分讲解。在编程中，我们可以将显示信息的代码移动到视图中，如同模型的构造函数一样。这样做可以达到以下效果：无需更改代码的其他部分，就可以按需切换输出类型，并且以多种方式显示数据。

在本章中，我们将在两个示例（健身应用程序和游戏 The Crypt）的环境中创建视图。对于健身应用程序，我们将构建一个简单版的控制台视图，然后使用 spacer 命名空间构建格式升级版本。对于 The Crypt，我们把原来在 Player 和 Place 构造函数中显示信息的代码分离出来，将其移动到新的玩家和场所视图代码中。

我们在本书第二部分一直追求代码模块化，看似是一番痛苦的煎熬，但模块化能够使简单的代码块在各个项目中自由穿梭，提高更新功能的灵活性。这些灵活便利性让我们感到欣慰。模块化的美丽前景正在向我们招手！

15.1　健身应用程序——显示最新的用户数据

我所在的团队正在编写一个多平台健身应用程序，让用户记录他们的锻炼情况。构建该应用程序包括以下任务：

（1）以字符串形式检索用户数据。

（2）将用户数据转换为用户模型。

（3）显示用户数据。

（4）为用户提供添加会话的接口。

我们已经设法完成了任务（1）和任务（2），将用户数据从简单的 JavaScript 对象转换为具有附加功能的用户模型，包括了 addSession 和 getData 方法（参见第 14 章）。下面，我们

来完成任务（3），即显示用户数据。

我们将创建两个视图，用于接收用户模型代码和显示模型中的数据代码。图 15-1 显示了不同的视图如何使用同一个模型来产生不同的输出。

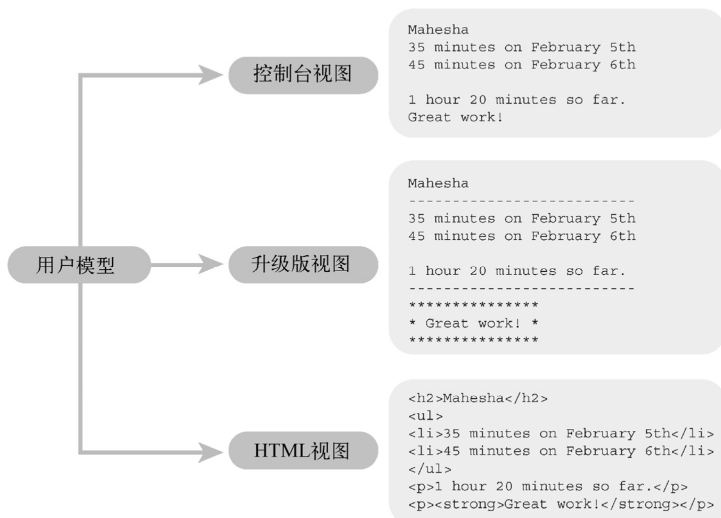


图 15-1　不同视图使用同一用户模型产生不同的输出 265

目前，我们一直在使用控制台视图，在本书的第三部分，我们将学习生成 HTML 视图，以便在网页上输出。

15.1.1　创建第一个健身应用视图

健身应用程序的第一个视图简洁明了，在控制台上显示用户信息，如下所示：

```
Mahesha
35 minutes on 2017-02-05
45 minutes on 2017-02-06
80 minutes so far
Well done!
```

要生成以上输出，需要一个视图和一个用户模型：

```
var user = new User("Mahesha");          //创建一个用户模型
user.addSession("2017-02-05", 35);
user.addSession("2017-02-06", 45);
userView.render(user);                   //将模型传递到视图进行显示
```

在代码清单 15-1 中，定义并立即执行创建用户视图的函数，返回一个简单的接口对象，并将接口分配给 userView 变量。要测试该视图，请调用 userView.render，就会生成上述显示。



代码清单 15-1　一个简单的用户视图
(http://jsbin.com/goqinep/edit?js,console)

```
var User = function (name) { /* 与第 14 章相同，JS Bin 上有显示 */ };
    var userView = (function () {        //使用函数表达式创建局部作用域
```

```
        var getInfo=function (userData){ //定义函数，从用户数据构建一个信息字符串
            var infoString = "\n" + userData.name + "\n";
            userData.sessions.forEach(function (session) {
                infoString += session.duration + " minutes on ";
                infoString += session.sessionDate + "\n";
            });
            infoString += "\n" + userData.total + " minutes so far";
            infoString += "\nWell done!\n";
            return infoString;
        };
        var render = function (user) { //定义 render 函数来输出用户模型的信息
            console.log(getInfo(user.getData()));
267     };
        return {        //返回可以访问 render 函数的简单接口对象
            render: render
        };
    })();               //立即执行圆括号内的函数表达式
    var user = new User("Mahesha");    //创建用户模型并添加两个会话
    user.addSession("2017-02-05", 35);
    user.addSession("2017-02-06", 45);
    userView.render(user);             //显示用户信息
```

以上代码中，将视图代码包裹在立即执行函数表达式中来创建局部作用域，这样做可以隐藏实施细节，也不需要额外的全局变量（有关 IIFE 和全局污染危险的更多信息，请参见第 13 章）。

代码的开头部分是在第 14 章出现的用户模型 User 构造函数。用户模型提供了 getData 方法，该方法返回一个简单的 JavaScript 对象：

```
{
    "name" : "Mahesha",
    "total" : 80,
    "sessions" : [
        { "sessionDate" : "2017-02-05", "duration" : 35 },
        { "sessionDate" : "2017-02-06", "duration" : 45 }
    ]
}
```

视图中的 getInfo 函数使用该数据对象来构建代码清单 15-1 前面显示的信息字符串。

如果我们面前呈现了两个视图，我们会依据什么标准而选择其中之一？

15.1.2 使用模块切换健身应用视图

最终，我会向团队成员展示一些视图，请他们从中选择最适合健身应用使用的产品。我的第二个视图中使用了 spacer 命名空间中的格式化函数向输出信息添加边框和方框。该函数的名称为 fitnessApp.userViewEnhanced。读者可以在 JS Bin 上(http://jsbin.com/puculi/edit?js,console)看到该视图的代码模块，输出如下所示：

```
Mahesha
----------------------------
35 minutes on 2017-02-05
45 minutes on 2017-02-06
80 minutes so far
----------------------------
**************
* Well done! *
**************
```

267

（第二个视图的代码与第一个视图的代码类似，因为我们的关注点是如何使用多个视图，所以本书中省略了第二个视图的代码。）

以下代码清单 15-2 显示了用于导入 spacer 和 fitnessApp 模块的 HTML script 元素。请注意，需要导入两个视图模块。



代码清单 15-2　测试两个用户视图（HTML）(http://jsbin.com/vamuzu/edit?html,js,console)

```
<!-- spacer -->
<script src="http://output.jsbin.com/juneqo.js"></script>
<!-- fitnessApp.User -->
<script src="http://output.jsbin.com/fasebo.js"></script>
<!-- fitnessApp.userView -->
<script src="http://output.jsbin.com/yapahe.js"></script>
<!-- fitnessApp.userViewEnhanced -->
<script src="http://output.jsbin.com/puculi.js"></script>
```

以下代码清单 15-3 显示了 JavaScript 代码，用于测试我们所创建的两个视图。



代码清单 15-3　测试两个用户视图 (http://jsbin.com/vamuzu/edit?html,js,console)

```
var user = new fitnessApp.User("Mahesha");    //创建用户模型并添加两个会话
user.addSession("2017-02-05", 35);
user.addSession("2017-02-06", 45);
fitnessApp.userView.render(user);             //测试第一个用户视图
fitnessApp.userViewEnhanced.render(user);     //测试升级版用户视图
```

多么可爱的视图！两个视图都运行良好，它们使用的数据来自同一用户模型，因此无论团队最终选择其中的哪一个都没有问题。团队中的其他人也可以根据需要很容易地创建更多的视图（Web 视图一定也很棒），而无需涉及用户模型代码。

15.1.3　下一个目标

我们已经学会从数据创建用户模型，并以多种方式使用视图显示该数据。下一个目标是给用户提供一个简单的方法，记录他们在控制台上的锻炼会话，并根据数据的变化获得更新视图、重新显示。本书将在第 16 章中介绍如何创建一个健身应用控制器来实现以上目标。

15.2 The Crypt——把显示代码从 Player 和 Place 模型中分离出来

在上一节中，我们学习了如何从头开始创建视图。首先有一个提供数据的用户模型，然后编写视图代码来显示数据。在本节中，我们将使用 The Crypt 的现有模型（Player 和
268 Place）。目前，显示模型数据的代码与模型本身混合在一起，图 15-2 显示游戏当前版本和新版本中的所有模块。为了方便日后更改，需要将显示代码与 Player 和 Place 模型分开，创建两个视图 playerView 和 placeView。

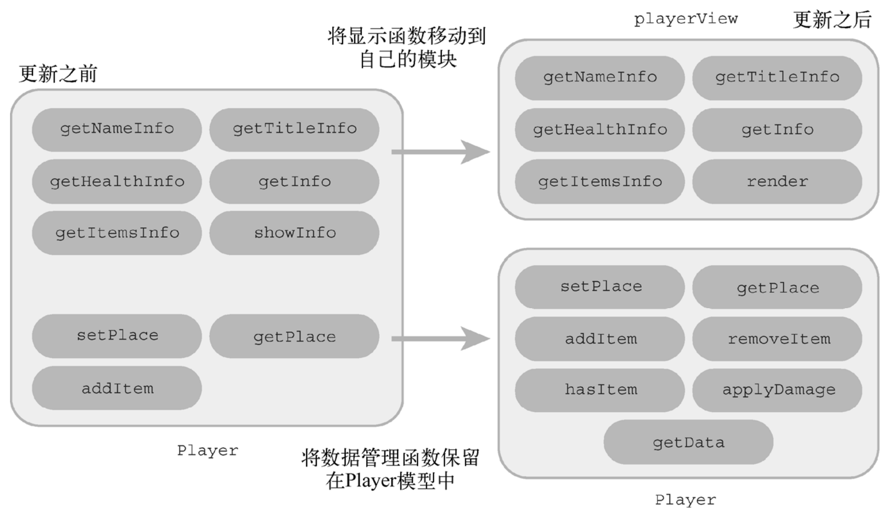
我们首先更新玩家模型，然后更新场所模型。



图 15-2　把 Player 和 Place 模型中的视图函数移至视图自己的模块中

15.2.1 为玩家创建视图

图 15-3 的左边是未更改之前，在 Player 模型中定义的函数。如图所示，该模型包括用于显示玩家信息的 6 个函数（在左上方），但是我们希望该模型只关注对玩家数据的管理，而不关注对数据的显示。图 15-3 的右边显示了更新的玩家模型，包括一些要添加的额外方法以及名为 playerView 的新模块，该模块只关注对玩家信息的显示。现在，我们实际上创建了两个模块：playerView 和 Player。



269 图 15-3　将显示函数从 Player 模型移动到自己的模块中

1．模型

模型不仅包含数据，而且提供用于管理数据的方法，例如添加、删除和更新数据，模型还可以防止对数据的直接访问。要创建一个玩家模型，可以使用 new 关键字调用 Player 构造函数，如下所示：

```
var player = new Player("Jahver", 80);
```

以下代码清单 15-4 显示了新版 Player 构造函数。虽然我们不想让 Player 模型费神去显示数据，但必须确保该数据可以被任何需要它的视图所用。我们仍然希望保护私人数据不被随意调整，所以创建了一个方法 getData，用来返回数据的副本。以下为返回的数据：

```
{
   "name" : "Jahver",
   "health" : 80,
   "items" : [ "a rusty key" ],
   "place" : "The Crypt"
}
```

为了处理第 16 章中的挑战，我们需要向 Player 构造函数添加三个方法：hasItem、removeItem 和 applyDamage。前往 JS Bin 并运行新的构造函数代码，可以在控制台提示符下执行以下操作（对于 undefined 的响应已省略）：

创建一个玩家，添加几个物品，并获得玩家的数据：

```
> var p = new theCrypt.Player("Dax", 10)
> p.addItem("a key")
> p.addItem("a lamp")
> p.getData()
[object Object] {
   health: 10,
   items: ["a key", "a lamp"],
   name: "Dax"
}
```

使用新的 item 方法来检查该玩家是否得到或删除某物品：

```
> p.hasItem("a key")
true
> p.hasItem("a sword")
false
> p.removeItem("a key")
> p.getData().items
["a lamp"]
```

对玩家施加伤害，然后查看他们的健康值： 270

```
> p.applyDamage(2)
> p.getData().health
8
```

```
> p.applyDamage(10)
> p.getData().health
-2
```



代码清单 15-4　Player 构造函数 (http://jsbin.com/yaneye/edit?js,console)

```
(function () {
  var Player = function (name, health) {
    var items = [];
    var place = null;
    this.addItem = function (item) {
      items.push(item);
    };
    this.hasItem = function (item){     //如果指定的物品在 items 数组中，则
                                        //返回 true
      return items.indexOf(item) !== -1;
    };
    this.removeItem = function(item){      //从 items 数组中删除指定的物品
      var itemIndex = items.indexOf(item);
      if (itemIndex !== -1) {
        items.splice(itemIndex, 1);
      }
    };
    this.setPlace = function (destination) {
      place = destination;
    };
    this.getPlace = function () {
      return place;
    };
    this.applyDamage = function (damage){  //从玩家的健康值中减去指定的
                                           //伤害值
      health = health - damage;
    };
    this.getData = function () {        //定义一个方法来返回模型数据的副本
      var data = {
        "name" : name,
        "health" : health,
        "items" : items.slice()         //使用没有参数的 slice 方法复制
                                        //items 数组
      };
      if (place !== null) {             //（如果玩家的位置已赋值）包括玩家所
                                        //在位置的标题
        data.place = place.title;
      }
      return data;                      //返回数据对象
    };
```

271

```
    };
    if (window.theCrypt === undefined) {
        window.theCrypt = {};
    }
    theCrypt.Player = Player;
})();
```

Player 构造函数不再为它所创建的模型分配任何显示方法。从先前版本中留下来的方法 addItem、setPlace 和 getPlace 仅仅用于管理模型中的玩家数据。用于显示的方法已经移至新的视图对象。

在以上代码中，hasItem 和 removeItem 都使用了 indexOf 数组方法。如果指定的物品在 items 数组中，indexOf 将返回该物品的索引值。如果该物品不在数组中，indexOf 将返回-1，如下所示：

```
["a key", "a lamp"].indexOf("a lamp");  // 返回值 1 --> 数组中的第二项
["a key", "a lamp"].indexOf("a sword"); // 返回值-1 --> 不在数组中
```

removeItem 方法使用 splice 从 items 数组中删除一个物品：

```
items.splice(itemIndex, 1);
```

我们知道，splice 的第一个参数是数组中要开始删除元素的索引值，第二个参数是要删除元素的数目。在 removeItem 中，我们需要删除单个指定的物品，因此 splice 的第二个参数为 1。

2. 视图

在以前我们编写的代码中，都是让玩家对象本身来生成显示内容。现在，我们使用玩家视图对显示内容进行重新创建。要显示一位玩家，可以调用视图的 render 方法，并将玩家模型作为参数传递给该方法：

```
theCrypt.playerView.render(kandra);
```

render 方法调用 getInfo 函数构建一个玩家信息字符串，并将返回的字符串显示在控制台上，如下所示：

```
****************************************
* Kandra (50)                          *
****************************************
Items:
- a rusty key
- a piece of cheese
****************************************
```

以下代码清单 15-5 显示了 playerView 代码，分配给 theCrypt.playerView 的接口带有方法 render。 272



代码清单 15-5　玩家视图
(http://jsbin.com/zucifu/edit?js,console)

```
(function () {
```

```
        var getNameInfo = function (playerData) {
            return playerData.name;
        };
        var getHealthInfo = function (playerData) {
            return "(" + playerData.health + ")";
        };
        var getItemsInfo = function (playerData) {
            var itemsString = "Items:" + spacer.newLine();
            playerData.items.forEach(function (item, i) {
                itemsString += " - " + item + spacer.newLine();
            });
            return itemsString;
        };
        var getTitleInfo = function (playerData) {
            return getNameInfo(playerData)+" "+getHealthInfo(playerData);
        };
        var getInfo = function (playerData) {   //使用玩家数据建立一个信息字符串
            var info = spacer.box(getTitleInfo(playerData), 40, "*");
            info += " " + getItemsInfo(playerData);
            info += spacer.line(40, "*");
            info += spacer.newLine();
            return info;
        };
        var render = function (player) {
            console.log(getInfo(player.getData()));//将玩家数据的副本传递给 getInfo
        };
        if (window.theCrypt === undefined) {
            window.theCrypt = {};
        }
        theCrypt.playerView = {        //将 render 方法设置为 playerView 的属性
            render: render
        };
    })();
```

render 是唯一给用户提供输出的函数，将生成的玩家信息字符串输出到控制台上。在该函数中，把玩家的数据传递给 getInfo。

现在，我们已经成功地将玩家模型从玩家视图中拆分出来。在第 17 章将看到，当我们
273 不想在控制台上显示玩家信息，而是要在网页上显示玩家信息时，只需要轻轻松松地更改一下视图就可以了。现在，我们先按照以上创建玩家的步骤，为游戏中的场所创建一个模型和一个视图。

15.2.2 创建场所视图

正如上一节中创建玩家那样，我们需要为场所重新编写构造函数，以便创建一个能保存每个地方数据的模型，并提供一些操作数据的方法。该模型也带有一个 getData 方法，以便视图可以获得每个地方的数据副本以供显示。然后，我们也需要创建一个视图用来在控制台

上输出位置数据。图 15-4 显示了如何将旧的模型代码拆分成为新的模型代码。

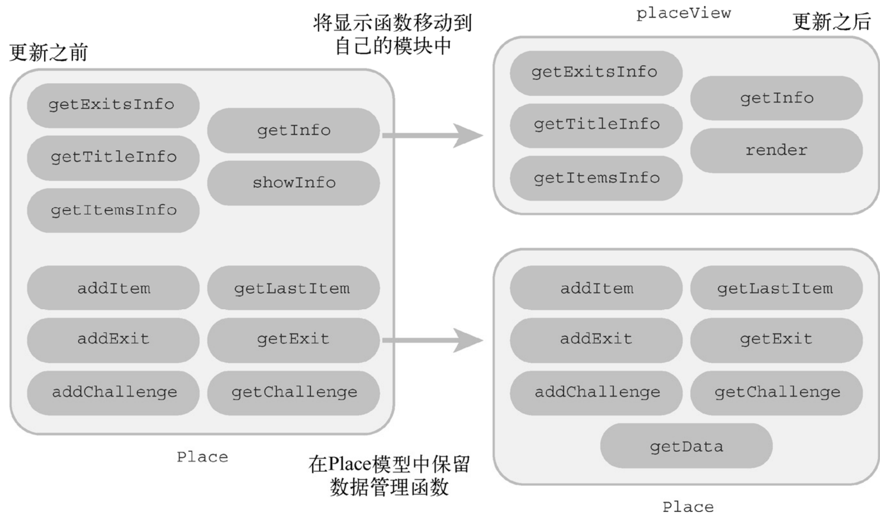


图 15-4　将显示函数从 Place 模型移动到自己的模块中

1. 模型

代码清单 15-6 显示了简化版的 Place 构造函数，已删除其显示代码，并且添加了一个 getData 方法。getData 返回的数据如下所示：

```
{
   "title" : "The Old Library",
   "description" : "You are in a dusty library. Books line the walls.",
   "items" : [ "a rusty key" ],
   "exits" : [ "west", "up" ]
}
```

其他方法与旧版的 Place 构造函数相同。 274



代码清单 15-6　简化版 Place 构造函数 (http://jsbin.com/vuwave/edit?js,console)

```
(function () {
   var Place = function (title, description) {
      var exits = {};
      var items = [];
      var challenges = {};
      this.addItem = function (item) {
         items.push(item);
      };
      this.getLastItem = function () {
         return items.pop();
      };
      this.addExit = function (direction, exit) {
```

```
                exits[direction] = exit;
            };
            this.getExit = function (direction) {
                return exits[direction];
            };
            this.addChallenge = function (direction, challenge) {
                challenges[direction] = challenge;
            };
            this.getChallenge = function (direction) {
                return challenges[direction];
            };
            this.getData = function () {        //定义一种方法以返回地点数据的副本
                var data = {
                    "title" : title,
                    "description" : description,
                    "items" : items.slice(),
                    "exits" : Object.keys(exits) //使用 Object.keys 将 direction
                                                 //数组分配给 exits 属性
                };
                return data;                     //返回数据对象
            };
        };
        if (window.theCrypt === undefined) {
            window.theCrypt = {};
        }
        theCrypt.Place = Place;
    })();
```

游戏中的每个场所模型都与某个目的地相连，而目的地也是地点模型，存储在地点模型的 exits 对象中，该对象的键是目的地的方向，例如，exits ["south"]是一个地点模型，其 title
275 属性为“The Old Library”。Object.keys 方法将对象的所有键作为数组返回，因此，Object.keys(exits)返回当前位置出口的所有方向，例如["south","east","west"]。

Place 构造函数中的 getData 方法可以返回某个地方的数据副本，包括使用 Object.keys(exits)生成的出口方向数组。视图将使用该数组来显示可用的出口，玩家可以从这些出口中做出选择，继续他们的冒险。

2. 视图

以下代码清单 15-7 是场所视图模块的代码，其工作原理与玩家视图模块相同，读者应该很熟悉了，该视图模块在控制台上的输出如下所示：

```
===============
= The Kitchen =
===============
You are in a kitchen. There is a disturbing smell.
Items:
- a piece of cheese
```

```
Exits from The Kitchen:
- east
- south
========================================
```



代码清单 15-7 场所视图 (http://jsbin.com/royine/edit?js,console)

```
(function () {
    var getItemsInfo = function (placeData){ //将该场所的数据传递到辅助函数
        var itemsString = "Items: " + spacer.newLine();
        placeData.items.forEach(function (item) {
            itemsString += " - " + item;
            itemsString += spacer.newLine();
        });
        return itemsString;
    };
    var getExitsInfo = function (placeData) {
        var exitsString = "Exits from " + placeData.title;
        exitsString += ":" + spacer.newLine();
        placeData.exits.forEach(function (direction) {
            exitsString += " - " + direction;
            exitsString += spacer.newLine();
        });
        return exitsString;
    };
    var getTitleInfo = function (placeData) {
        return spacer.box(placeData.title,placeData.title.length + 4, "=");
    };
    var getInfo = function (placeData) { //将该场所的数据传递给主函数 getInfo
        var infoString = getTitleInfo(placeData);
        infoString += placeData.description;
        infoString += spacer.newLine() + spacer.newLine();
        infoString += getItemsInfo(placeData) + spacer.newLine();
        infoString += getExitsInfo(placeData);
        infoString += spacer.line(40, "=") + spacer.newLine();
        return infoString;
    };
    var render = function (place) {
        console.log(getInfo(place.getData()));   //调用 getData 获取场所模型
                                                 //数据的副本
    };
    if (window.theCrypt === undefined) {
        window.theCrypt = {};
    }
    theCrypt.placeView = {
        render: render
```

276

```
    };
})();
```

如同玩家的视图模块一样，场所的视图模块从主要显示函数 getInfo 调用一些辅助函数：getItemsInfo、getExitsInfo 和 getTitleInfo，以建立关于模型的信息字符串。render 函数是唯一可以产生用户可见输出的函数，该函数在控制台上显示组合好的场所信息。

我们现在已经拥有玩家和场所的模型构造函数和视图，将数据和数据的显示进行了有效分离：模型只关注数据的操作，与数据显示无关；视图只关注显示数据，与数据更改无关。

以上两个视图中都使用了 render 方法来显示模型：

```
// 创建模型
var kandra = new Player("Kandra", 50);
var library = new Place("The Library", "You are in a dusty library.");
// 使用视图显示模型信息
playerView.render(kandra);
placeView.render(library);
```

除了显示玩家和场所的信息之外，我们还需要在用户玩游戏时给玩家一些提示消息，例
277 如，告诉他们是被僵尸咬伤了，还是被豹子撕伤了。

15.3 与玩家交互——消息视图

当冒险者在游戏中行进时，需要让他们知道周围正在发生什么情况。我们使用玩家视图为玩家更新健康值和他们携带的物品，使用场所视图显示每个地方的标题和说明，并列出此地的物品和出口。在游戏中，当玩家尝试了一次无效的行动或遭到伤害时，还需要一种能够为他们显示反馈信息的方法，如下所示：

```
> game.go("north")
*** There is no exit in that direction ***
> game.get("a piece of cheese")
*** That item is not here ***
> game.use("a lamp north")
*** That doesn't help. The door is still locked. ***
```

为了处理此类消息，我们需要创建一个消息视图。如同其他视图一样，在其公共接口处有一个 render 方法，游戏将要显示的文本传递给 render 方法：

```
theCrypt.messageView.render("That item is not here.");
```

以下代码清单 15-8 显示该视图的代码。局部 getMessageInfo 函数返回一个供显示的字符串，render 函数将它在控制台上输出。



代码清单 15-8　消息视图
(http://jsbin.com/jatofe/edit?js,console)

```
(function () {
```

```
    var getMessageInfo = function (messageData) {   //编排消息的格式使该消
                                                    //息看上去很紧急
        return "*** " + messageData + " ***";
    };
    var render = function (message) {
        console.error(getMessageInfo(message));  //使用 console.error 来提
                                                 //示消息，提示消息的形式与
                                                 //报错消息一样醒目
    };
    if (window.theCrypt === undefined) {
        window.theCrypt = {};
    }
    theCrypt.messageView = {
        render: render
    };
})();
```

在以上代码中，render 函数使用了 console.error 方法，类似于 console.log。但是在日常编程中，开发人员一般使用 console.error 来提示程序中的错误，控制台在显示错误时通常与显示普通消息不同，错误通常显示为红色。这刚好满足了我们的需要：在控制台上醒目地向玩家提示消息。 278

在本书的第 14 章，我们为 The Crypt 地图数据添加了挑战。如何使用消息视图来显示与每个挑战相关的成功或失败消息？我们首先需要一些代码来检查用户应对挑战的操作，并基于此来更新玩家和场所模型，然后使用视图来显示游戏的最新状态。我们现在需要控制器，相关内容将在第 16 章进行讨论。

15.4　本章小结

- 创建视图来显示模型中的数据。
- 模型和视图分别关注不同的内容：模型操作数据，视图显示数据。
- 创建多个视图，以不同方式显示相同的模型数据。
- 保持所有视图的接口一致。例如，本章中所有视图都用 render 方法作为接口，调用方式也相同：

```
fitnessApp.userView.render(user);
fitnessApp.userViewEnhanced.render(user);
theCrypt.playerView.render(player);
theCrypt.placeView.render(place);
theCrypt.messageView.render(message);
```

279

第 16 章　控制器：连接模型和视图

本章内容包括:

- 根据用户输入执行操作
- 响应用户操作更新模型
- 将更新的模型传递到视图进行显示
- 完成控制台版本的游戏 The Crypt

本书的第二部分一直致力于组织代码。随着程序变得越来越长，我们逐渐体会到前期对代码费心费力的组织工作所带来的便利之处。组织代码使我们更易于专注程序的单个部分，
280 更易于切换模块以更改某些功能，还更易于在多个项目中重用代码。

得益于将程序分解成模块的做法，现在我们可以为每个模块指定一个特定的任务。第 14、15 和 16 章构成三部曲，每章着眼于一个常见任务，每个任务由一个模块来执行。在第 14 章介绍了模型；在第 15 章介绍了视图；现在我们用控制器将以上二者联系起来。

为了查看操作中的控制器，并了解控制器是如何根据用户输入来管理模型和更新视图的，我们将继续在健身应用程序和游戏 The Crypt 两个项目中进行工作。在健身应用程序中，用户将记录锻炼的会话；在游戏 The Crypt 中，当玩家探索危险的地方时，将面临难题需要解决。他们能够在健康值跌到零之前成功逃脱吗？

16.1　构建一个健身应用控制器

团队成员已经向亲朋好友推荐了我们正在编写的健身应用程序。现在有一群人在排队等着测试该程序。他们是一个运动发烧友群体，并一直热衷于把锻炼情况记录在纸上。我们的工作目前进展顺利，下一步需要做什么？以下是团队提出的要求：

（1）以字符串形式检索用户数据。

（2）将用户数据转换为用户模型。

（3）显示用户数据。

（4）提供一个用户添加会话的接口。

对照以上任务清单，我们在第 14 章创建了模型，已经完成了任务（2）；在第 15 章创建了视图，已经完成了任务（3）；任务（1）涉及在互联网上检索数据，读者将在本书的第三部分进行学习；现在，我们就来着手完成任务（4）。我们需要为那些狂热的健身爱好者提供一个简单的方式来记录他们所完成的每一次运动。

我们打算用一条命令来记录用户的运动情况：

```
app.log("2017-02-07", 50)
```

用户可以调用 app.log 方法，传递他们的运动日期和每次锻炼的持续时间（以分钟为单位）。该命令如何能够到达用户模型？视图如何能够相应地更新显示？这就是控制器要做的工作。

16.1.1　控制器的作用

控制器对其他程序的作用犹如把各个音乐曲目编排成优美和谐的管弦乐。控制器针对用户输入做出反应，然后更新模型和视图。图 16-1 显示了健身应用程序在 JS Bin 中打开时会加载的模块。我们已经拥有第 14 章和第 15 章的数据，拥有 User 构造函数和视图，现在需要创建一个控制器来处理各个部分之间的交互。

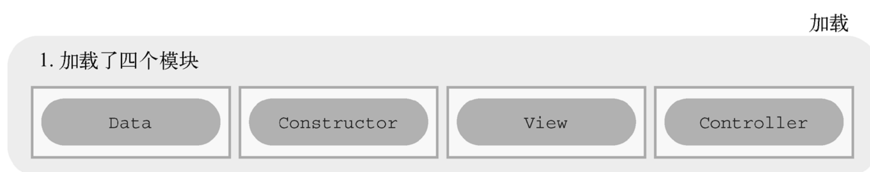


图 16-1　健身应用程序的四个模块

图 16-2 显示了控制器如何参与初始化应用程序。控制器使用 User 构造函数，从数据构建用户模型。当应用程序运行时，控制器对用户调用 app.log 做出反应：将记录的会话添加到用户模型，然后将更新的模型传递到视图进行显示。

现在，我们大致明白了控制器需要做什么。具体将如何实现？ 281

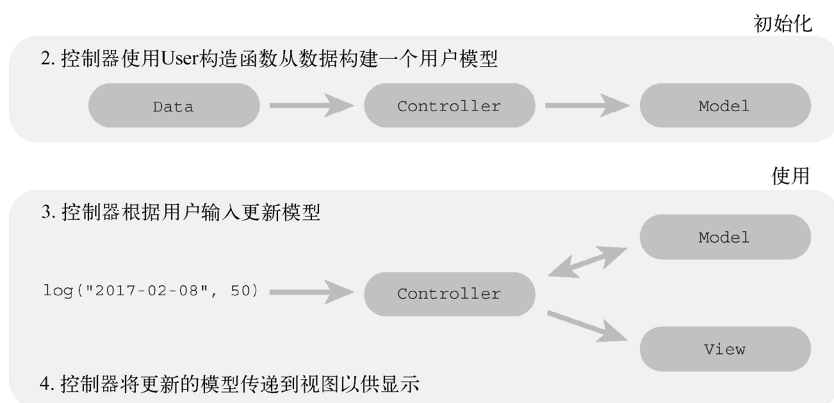


图 16-2　控制器在健身应用程序中执行的任务

16.1.2　构建一个健身应用控制器

以下为控制器中的重要语句：

```
var user = buildUser(userData);     //将用户数据转换为用户模型
> app.log("2017-02-08", 50)         //用户在记录一个会话
user.addSession("2017-02-08", 50); //控制器将其添加到模型中
fitnessApp.userView.render(user);  //然后更新视图
```

以下代码清单 16-1 显示了控制器的完整代码。



代码清单 16-1　健身应用程序控制器
(http://jsbin.com/goniro/edit?js,console)

```
(function () {
   var buildUser = function (userData){  //定义函数，从用户数据构建用户模型
      var user = new fitnessApp.User(userData.name);
      userData.sessions.forEach(function (session) {
         user.addSession(session.sessionDate, session.duration);
      });
      return user;
   };
   var init = function (userData) {  //定义 init 函数，对应用程序进行初始化
      var user = buildUser(userData);//从用户数据构建模型并将其分配给用户变量
      fitnessApp.userView.render(user);    //将新用户模型传递到视图进行显示
      return {        //返回一个具有单个方法 log 的接口
         log: function (sessionDate, duration) {
            user.addSession(sessionDate, duration);
            fitnessApp.userView.render(user);
            return "Thanks for logging your session.";
         }
      };
   };
   if (window.fitnessApp === undefined) {
      window.fitnessApp = {};
   }
   fitnessApp.init = init;    //通过将 init 方法添加到 fitnessApp 命名空间
                              //中，使 init 方法可用
})();
```

282

当程序加载控制器模块时，将 init 函数添加到 fitnessApp 命名空间。我们可以通过调用 init 并向其传递用户数据来启动该应用程序：

```
var app = fitnessApp.init(fitnessApp.userData);
```

init 函数返回一个具有单个方法 log 的接口。通过将返回的接口分配给 app，可以让用户调用 app.log 来记录其会话。

16.1.3　将各个部分组合在一起用于健身应用程序

代码清单 16-2 和 16-3 显示了 JS Bin 上的健身应用程序（HTML 和 JavaScript）。运行程序可以让用户记录锻炼情况，如下所示：

```
>app.log("2017-02-08", 55)
```

控制器将以上会话记录添加到用户模型，并将更新的模型传递给视图，输出如下所示：

```
Mahesha
```

```
35 minutes on 2017-02-05
45 minutes on 2017-02-06
55 minutes on 2017-02-08
135 minutes so far
Well done!
Thanks for logging your session.
```

在以下代码清单 16-2 中，使用 HTML 脚本元素加载健身应用程序（图 16-1）中的 4 个模块，再将其分配给 fitnessApp 命名空间共享其属性、对象和函数。 283



代码清单 16-2　健身应用程序（HTML）
(http://jsbin.com/huxuti/edit?html,js,console)

```
<!--fitnessApp.userData -->
<script src="http://output.jsbin.com/tenuwis.js"></script>
<!--fitnessApp.userView -->
<script src="http://output.jsbin.com/yapahe.js"></script>
<!--fitnessApp.User -->
<script src="http://output.jsbin.com/fasebo.js"></script>
<!--fitnessApp.controller -->
<script src="http://output.jsbin.com/goniro.js"></script>
```

以下代码清单 16-3 显示了初始化程序所需的一行 JavaScript 代码，该代码提供了可供用户使用的 app.log 方法。



代码清单 16-3　健身应用程序
(http://jsbin.com/huxuti/edit?js,console)

```
var app = fitnessApp.init(fitnessApp.userData);
```

运行以上程序，就可以添加更多记录运动情况的会话。

16.1.4　下一个目标

以上健身应用程序的下一个目标就是团队提出的第（4）项任务：从互联网获取用户数据作为文本，然后将该文本转换为 JavaScript 对象，再传递到 fitnessApp.init。在第 20 章，读者将重新回到这个应用程序，继续完善它，并完成最后一项任务。

16.2　游戏 The Crypt——添加控制器

我们继续完善本书第二部分中的游戏 The Crypt。在本章结束时，我们将完成一个能够运行的控制台应用程序。在这个程序中，有一些挑战需要玩家去应对，还有一些危险需要玩家去注意：当玩家的健康值下降到零时，游戏就会结束。我们的目标是让游戏完全运行起来，现在游戏拼图板只缺少一块儿内容：控制器。添加上控制器，就大功告成了！

现在，我们已经拥有游戏 The Crypt 的数据、模型和视图，控制器是将上述所有部分联系在一起的模块。控制器将地图数据传送到地图构建器，并将模型数据传递给视图。控制器

为用户提供了一个玩游戏的接口，该接口还能够对玩家命令做出响应。下面，我们就来学习这个身兼数职的控制器，看看它如何与游戏中的其他模块相互衔接、相互适应。

图 16-3 显示了构成游戏 The Crypt 的所有模块。控制器模块已经替换了以前标记为“Game init”的模块。既然我们正在使用模型和视图，就应该把这个模块称作一个控制器。
284 读者不禁会想，这是为什么？ 控制器又能够做什么？

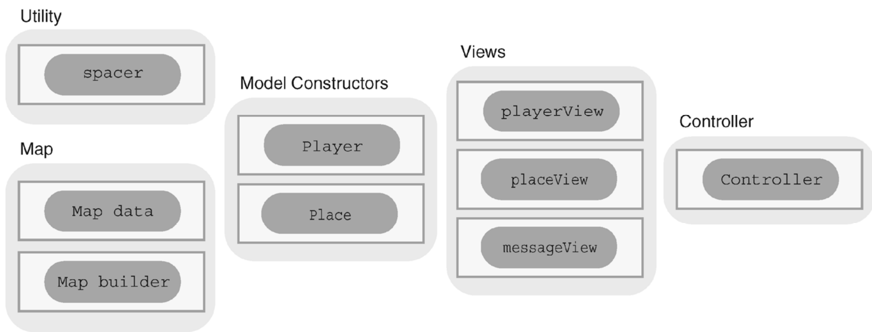


图 16-3　构成游戏 The Crypt 的各个模块

16.2.1　控制器在游戏中的作用

控制器能够对游戏进行初始化，并在游戏运行时对用户的输入做出响应：根据玩家的操作来更新相应的模型和视图。

1. 对游戏进行初始化

当游戏首次加载时，控制器将：

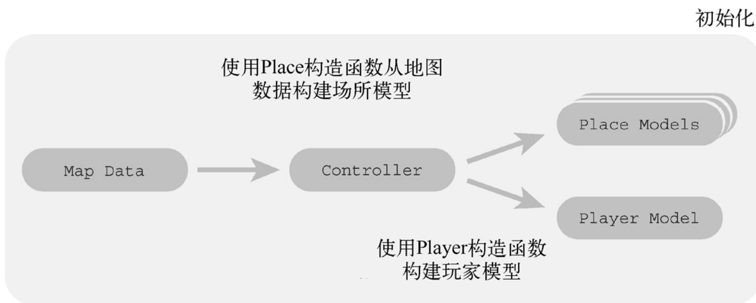
（1）使用 Place 构造函数从地图数据构建场所模型。

（2）使用 Player 构造函数构建玩家模型。

（3）将地图数据中指定的第一个位置分配为玩家位置。

（4）提供 UI。

步骤（1）和步骤（2）如图 16-4 所示。



285 图 16-4　当游戏加载时，控制器就会构建玩家和场所模型

2. 对用户的输入做出响应

在游戏运行的过程中，控制器将：

（1）检查玩家的操作是否有效。

（2）更新玩家和场所模型。

（3）将更新的模型传递到视图进行显示。

（4）将反馈消息传递到消息视图。

（5）如果玩家的健康状况降到零，则终止游戏。

以上步骤（2）～（4）如图 16-5 所示。

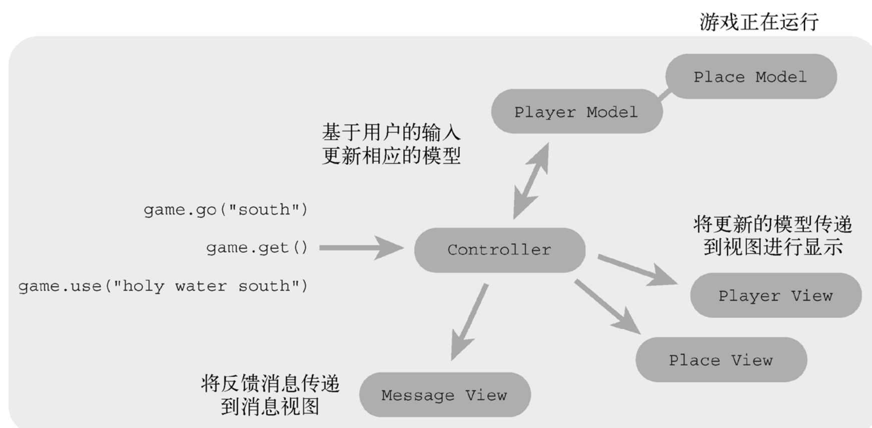


图 16-5　控制器响应用户操作并更新模型和视图

玩家模型包括一个 getPlace 方法，控制器使用该方法访问玩家的当前位置（请注意，在图 16-5 中 Place 模型如何连接到 Player 模型）。

16.2.2　走近控制器代码

我们已经大致了解控制器在游戏 The Crypt 中扮演的角色，现在准备进一步探索其运行代码。虽然读者以前曾经读过很多代码，但是面对一个较长的代码清单时，还是会望而却步。所以在本书中，将这次探索拆分成以下 4 节内容：

- (16.3) 控制器的整体结构
- (16.4) 游戏初始化，监控玩家的健康值，更新显示和结束游戏
- (16.5) 处理玩家的命令和挑战——get、go 和 use
- (16.6) 运行游戏

下面，让我们把游戏运行起来吧！ 286

16.3　The Crypt——控制器代码的结构

我们已经看到，在 The Crypt 中，控制器要执行许多任务：启动、监视和结束游戏。为了帮助读者了解如何将各部分整合为一个整体，以下代码清单 16-4 中省略了函数体，只关注代码中的变量。欲查看完整代码、函数体和所有变量，读者可以访问 JS Bin 上的链接网址。



代码清单 16-4　游戏控制器
(http://jsbin.com/yeqicu/edit?js)

```
(function () {
```

```
    var player;
    var inPlay = false;    //在游戏结束时使用 inPlay 进行标记
    var init = function (mapData, playerName) {     //构建地图，创建一个玩
                                                     //家，并开始游戏
        /*代码清单 16.5 */
    };
    var checkGameStatus = function(){/*16.6*/};     //检查玩家的健康值是否
                                                     //已降至零
    var render = function () { /* 16.7 */ };        //调用视图以更新显示
    var renderMessage = function (message) {/*16.7*/};
    var get = function () { /* 16.8 */ };           //响应用户的命令
    var go = function (direction) { /* 16.9 */ };
    var use = function (item, direction) {/*16.10*/};
    window.game = {        //将接口分配给全局变量 game
        get: get,
        go: go,
        use: use,
        init: init
    };
})();
```

以上代码清单 16-4 中包括一些注释，帮助读者查看在本章中省略的函数体代码。

现在，我们可以开始游戏！终止游戏！监控玩家的健康值！更新显示！（对不起，我太兴奋了，游戏的完整版本指日可待！）

16.4　The Crypt——开始和终止游戏

虽然在 get、go 和 use 函数中已包含游戏的主要动作，但是我们还是需要能够便捷地开始和终止游戏，并始终显示更新后的最新信息。在游戏中，玩家面临残酷的挑战，应对挑战
287 可能会耗尽他们的健康值，这时就需要查看他们是否依然强健和热血，足够继续游戏。

16.4.1　游戏初始化

要开始游戏，需要调用 init 方法进行初始化：

```
game.init(map, playerName);
```

init 方法可以构建地图、创建玩家模型、将玩家的位置设置到正确的地方，然后显示玩家和地点信息，如以下代码清单所示。



代码清单 16-5　init 函数
(http://jsbin.com/yeqicu/edit?js)

```
var player;
var inPlay = false;    //声明一个变量来记录游戏是否正在运行
var init = function (mapData, playerName) {         //构建地图，存储起始位置
```

```
        var firstPlace = theCrypt.buildMap(mapData);
        player = new theCrypt.Player(playerName, 50);   //设置玩家
        player.addItem("The Sword of Doom");
        player.setPlace(firstPlace);
        inPlay = true;          //更新 inPlay：游戏正在运行
        render();               //显示地点和玩家信息
    };
```

以上代码中，在 init 函数之外声明了 player 和 inPlay 变量，以供控制器中的其他函数访问。buildMap 函数返回游戏中的起始位置，并将该位置设置为玩家的当前位置。init 函数调用 render 来显示起始位置和玩家信息。

render、get、go 和 use 函数都使用了 inPlay 变量。只有在 inPlay 为 true 时才执行这些常规任务；当玩家的健康值下降到零时，控制器将 inPlay 设置为 false，上述任务全都无法执行。因此，非常有必要关注玩家的健康值。

16.4.2　监测玩家的健康值

每当玩家受到伤害时，都需要检查他们的健康值是否已经降到零。如果健康值降到零，就意味着游戏结束！控制器中的 checkGameStatus 函数就在执行这项检查：如果玩家已经被僵尸咬伤，被豹子撕伤以及遭遇讨厌的碎片，则将 inPlay 变量设置为 false。如以下代码清单所示。 288



代码清单 16-6　检查健康值是否已降至零
(http://jsbin.com/yeqicu/edit?js)

```
var checkGameStatus = function () {
    if (player.getData().health <= 0) {  //使用 getData 获取玩家的数据并检查
                                          //其健康状况
        inPlay = false;       //通过将 inPlay 设置为 false 停止游戏
        renderMessage("Overcome by your wounds...");
        renderMessage("...you fall to the ground.");
        renderMessage("- Your adventure is over -");
    }
};
```

在以上代码中，条件语句使用了比较运算符<=（小于或等于），用来检查健康值是否小于或等于零：

```
player.getData().health <= 0
```

如果玩家的健康值小于或等于零则意味着玩家死亡，就会终止游戏并显示最后一条消息。如果玩家的健康值大于零，游戏还可以继续，checkGameStatus 也不进行任何操作。

16.4.3　更新显示——使用视图模块的函数

因为某一个视图只是在一个地方被引用，控制器并不直接调用视图模块的 render 方法，而是使用控制器自己的函数，这样更易于按需在不同的视图模块之间进行切换。代码清单如下所示。



代码清单 16-7　更新显示
(http://jsbin.com/yeqicu/edit?js)

```
var render = function () {
    console.clear();      //清除显示内容
    if (inPlay) {         //仅当游戏正在进行时，才会对场所和玩家的显示进行更新
        theCrypt.placeView.render(player.getPlace());
        theCrypt.playerView.render(player);
    }
};
var renderMessage = function (message) {
    theCrypt.messageView.render(message);
};
```

init 函数用于启动游戏，启动之后就可以在整个游戏过程中使用 checkGameStatus、render 和 renderMessage。游戏中玩家的主要动作来自 get、go 和 use 三个函数，当玩家根据所持物品、所在位置的出口以及所在位置的挑战做出决定采取行动时，会使用 go 和 use。下
289 面，我们就去处理玩家的命令——开始行动！

16.5　The Crypt——发布命令和解决难题

在 The Crypt 游戏探索中，用户只能进行为数不多的几项操作。控制器模块将以下接口分配给全局对象 window：

```
window.game = {
    get: get,
    go: go,
    use: use,
    init: init
};
```

用户在控制台上调用游戏控制器的接口方法，来进行游戏中的动作，例如 game.get()或 game.go("south")；然后，游戏控制器对用户的指令做出响应，去创建、访问或更新模型，并将更新后的模型传递到视图进行显示。

我们已经了解了 init 函数，本节将逐一分析 get、go 和 use 函数。get 函数没有多大变化，go 函数用来检查挑战，而 use 则是一个全新的函数。

16.5.1　使用 game.get 拾取物品

探索游戏的玩家从一个地方旅行到另一个地方时会发现一些物品，这些物品能够帮助他们应对即将面临的挑战。get 方法为玩家提供了一种拾取他们所发现物品的方法，如下所示。



代码清单 16-8　get 函数
(http://jsbin.com/yeqicu/edit?js)

```
var get = function () {
    if (inPlay) {     //仅当游戏仍处于进行状态时才能获取物品
        var place = player.getPlace();     //在玩家的当前位置检索最后一个物品
```

```
        var item = place.getLastItem();
        if (item !== undefined) {
            player.addItem(item);           //如果检索到物品，请将其添加到玩家列表
            render();                       //更新显示，表明该物品已移动
        } else {
            renderMessage("There is no item to get");
        }
    } else {
        renderMessage("The game is over!");      //如果 inPlay 为 false，让玩
                                                 //家知晓游戏结束
    }
    return "";
};
```
290

在以上程序中，将玩家拾取的物品从该地点的物品列表末尾删除，并将其添加到玩家的物品列表中。如果此地没有任何物品，getLastItem 将返回 undefined。

下面我们来分析 go 和 use 函数，二者都涉及对挑战进行检查。

16.5.2　列出挑战的各个属性

既然玩家喜欢挑战，就要体验一下危险的刺激。游戏刚开始，玩家对象就拥有健康属性，游戏开发者蓄意在玩家游走于迷宫之时对他们做些恶劣的伤害。

我们在第 14 章开发地图数据时，首次在游戏中看到了挑战，但是直到现在都没有真正让这些挑战派上用场。在游戏中玩家可以使用两个命令 go 和 use 来检查指定方向的那些挑战。千万不要忘记，作为一个对象，挑战必须拥有一些属性：

```
"challenge" : {
    "message" : "A zombie sinks its teeth into your neck.",
    "success" : "The zombie disintegrates into a puddle of goo.",
    "failure" : "The zombie is strangely resilient.",
    "requires" : "holy water",
    "itemConsumed" : true,
    "damage" : 20
}
```

表 16-1 列出了挑战对象的各个属性及其含义，以及某一属性是否为必需的属性。

表 16-1　挑战对象的属性

属性	含　义	是否必需
message	当玩家试图朝着某出口的方向行进时显示给玩家的消息，消息中会显示玩家尚未应对的挑战	是
success	应对某一挑战，玩家用对了东西，也就是说，玩家确实使用了所需物品，就会显示 success（成功）	是
failure	当玩家使用错误的物品来应对某一挑战时，就会显示 failure（失败）	是
requires	应对某一挑战所需的物品	是
itemConsumed	物品被使用后是否从玩家的物品列表中将其删除	否
damage	在玩家没有成功地应对挑战就试图向出口方向前行时，从玩家的健康值中减去的量	否
complete	表明某一挑战是否已经完成。在初始数据中，此项内容通常缺失；在游戏进行的过程中，当挑战得以解决时，其值为 true	否

291

要从一个场所对象中检索某一挑战，请先指定方向，调用 getChallenge：

```
place.getChallenge（"south"）;
```

如果在这个方向上没有挑战，getChallenge 会返回 undefined；如果有挑战，就需要积极应对。玩家真正应对挑战的第一个命令是 game.go。

16.5.3 使用 game.go 来移动位置

玩家使用控制器的 go 函数从一个地方移动到另一个地方。如果玩家在某个出口遇到一把锁，或者遇到一只豹子，就无法前行，这时，控制器需要让玩家知晓他们的行程受阻。当玩家指定前行方向时，可以在控制台上调用 go：

```
>game.go（"south"）
```

完整的 go 函数如下所示，在代码清单后面有详细解释。



代码清单 16-9　go 函数
(http://jsbin.com/yeqicu/edit?js)

```
var go = function (direction) {
    if (inPlay) {        //收集所需的信息
        var place = player.getPlace();
        var destination = place.getExit(direction);
        var challenge = place.getChallenge(direction);
        if (destination === undefined) {
            renderMessage("There is no exit in that direction");
        } else {
            if ((challenge === undefined) || challenge.complete) {
                player.setPlace(destination);    //如果此地没有挑战或挑战已完
                                                 //成，请移动玩家并更新显示
                render();
            } else {             //让玩家承受挑战失败所带来的伤害
                if (challenge.damage) {
                    player.applyDamage(challenge.damage);
                    }
                render();        //针对健康值的更改，更新显示
                renderMessage(challenge.message);     //显示初始挑战消息
                checkGameStatus();          //检查玩家的健康值是否已下降到零
            }
        }
    } else {
        renderMessage("The game is over!");
    }
    return "";
};
```

292

函数先收集所需信息以便根据玩家的请求采取行动：

```
var place = player.getPlace();
var destination = place.getExit(direction);
var challenge = place.getChallenge(direction);
```

也许在玩家指定的方向并没有出口，因此需要先核查：

```
if (destination === undefined) {
   renderMessage("There is no exit in that direction");
} else {
   // 核查是否存在挑战
}
```

例如，在指定的方向上确实有一个出口，但事实上出去之后是一堵墙，玩家当然不会走进一堵墙。如果此地没有挑战，或者玩家已经完成挑战，就可以将他们移动到下一个目的地：

```
if ((challenge === undefined) || challenge.complete) {
   player.setPlace(destination);
   render();
} else {
   //提到豹子
   //所带来的伤害
}
```

if 条件中的符号||是逻辑或运算符。逻辑或运算符同时检查由两个表达式组成的条件，如果第一个表达式 challenge===undefined 或第二个表达式 challenge.complete 的值为 true，则整个条件为 true；如果两个表达式都为 false，则整个条件为 false。与逻辑或运算符不同，JavaScript 中还有一个逻辑与运算符&&，如果两个表达式都为 true，则值为 true，否则为 false。

如果玩家没有完成某个挑战，就要承受挑战失败所带来的伤害（例如碰撞、瘀伤、豹子咬伤等）。

```
if (challenge.damage) {
   player.applyDamage(challenge.damage);
}
```

最后，更新显示健康值的变化和该挑战的初始消息，并查看玩家的健康值是否仍然高于零：如果高于零，就继续游戏。代码如下：

```
render();
renderMessage(challenge.message);
checkGameStatus();
```

以上都是玩家独自在冒险，并未遭遇外来攻击。假设前方有一只饥饿的豹子挡在道路上，该怎么办？玩家如何才能打败豹子？ 293

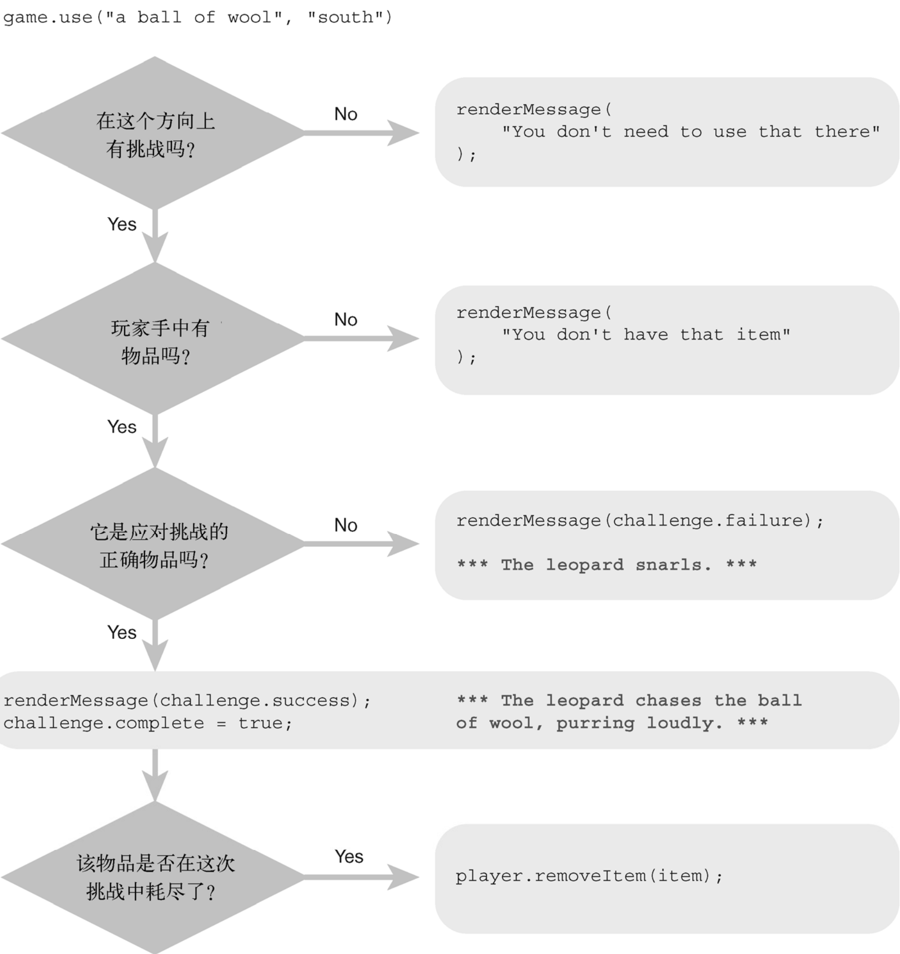
16.5.4　用 game.use 打败豹子

如果在前行的道路上遇到一把锁或者有豹子挡道，玩家可以调用控制器中的 use 函数来

应对挑战，指定要使用的物品和前行方向：

```
>game.use("a ball of wool", "south")
*** The leopard chases the ball of wool, purring loudly. ***
```

完整的 use 函数如代码清单 16-10 所示。程序中虽然有一些嵌套的 if 语句，但不必过分担心，重要节点如图 16-6 所示，读者可以顺着兔子洞下去探寻其工作原理。



294 图 16-6　use 函数中所做出的关键决策



代码清单 16-10　use 函数
(http://jsbin.com/yeqicu/edit?js)

```
var use = function (item, direction) {
    if (inPlay) {
        var place = player.getPlace();    //获取指定方向的挑战
        var challenge = place.getChallenge(direction);
        if ((challenge === undefined)||challenge.complete){  //检查挑战是否完
                                                              //成或该挑战不存在
```

```
        renderMessage("You don't need to use that there");
        } else {
            if (player.hasItem(item)) {          //检查玩家是否有指定物品
                if (item === challenge.requires) {
                    renderMessage(challenge.success);//显示成功消息并将该挑战标
                                                     //记为已完成
                    challenge.complete = true;
                    if (challenge.itemConsumed){ //如果物品在挑战中消耗，从玩
                                                 //家中删除该物品
                    player.removeItem(item);
                    }
            } else {
                        renderMessage(challenge.failure);   //如果玩家使用了
                                                            //错误的物品，则
                                                            //显示失败消息
            }
            } else {
                renderMessage("You don't have that item"); //让玩家知道他们
                                                            //并不拥有所需
                                                            //物品
            }
    } else {
        renderMessage("The game is over!");
    }
    return "";
};
```

在以上代码中，函数在开始处收集所需要的信息：

```
var place = player.getPlace();
var challenge = place.getChallenge(direction);
```

在玩家指定的方向有可能没有挑战，因此需要检查是否存在挑战：

```
if ((challenge === undefined) || challenge.complete) {
    renderMessage("You don't need to use that there");
} else {
    // 存在需要应对的挑战
}
```

295

有些玩家遮遮掩掩！因此需要检查他们是否拥有需要的物品。如果没有，就礼貌地提示他们：

```
if (player.hasItem(item)) {
    // 检查它是不是合适的物品
} else {
    renderMessage("You don't have that item");
}
```

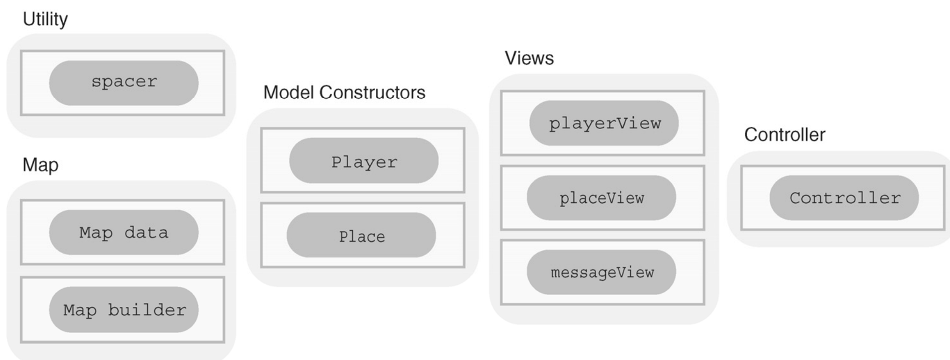
他们正在使用一个物品来应对一场实际的挑战。该物品是否为正确的应对物品？如果不是，请告诉他们：

```
if (item === challenge.requires) {
    // 完成挑战
} else {
    renderMessage(challenge.failure);
}
```

很好，我们通过使用 get、go 和 use 函数游历了仙境。这些函数本来并不太烦琐，但是当内部嵌套了太多层的 if 和 else 时，就可能会变得相当臃肿。歇歇脚，品品茶，犒劳犒劳自己，但千万不要迟到，耽误了第 16.6 节——那才是冒险真正开始的地方。

16.6 The Crypt——运行游戏

游戏运行需要我们创建的所有模块，然后调用 game.init 方法，将地图数据和玩家的名称传递给该方法。图 16-7 显示了所有涉及的模块。



296 图 16-7 游戏 The Crypt 中的各个模块

以下代码清单用于加载模块的 HTML 脚本元素。



代码清单 16-11 加载游戏模块（HTML）(http://jsbin.com/fociqo/edit?html,console)

```
<script>
console.log("Loading The Crypt ...");        //让玩家知道模块正在加载
</script>
<!-- spacer -->
<script src="http://output.jsbin.com/juneqo.js"></script>
<!-- Place 构造函数 -->
<script src="http://output.jsbin.com/vuwave.js"></script>
<!-- Player 构造函数-->
<script src="http://output.jsbin.com/nonari.js"></script>
<!--玩家视图 -->
```

```
<script src="http://output.jsbin.com/zucifu.js"></script>
<!--场所视图-->
<script src="http://output.jsbin.com/royine.js"></script>
<!--消息视图-->
<script src="http://output.jsbin.com/jatofe.js"></script>
<!--地图数据 -->
<script src="http://output.jsbin.com/hozefe.js"></script>
<!--地图构造器 -->
<script src="http://output.jsbin.com/paqihi.js"></script>
<!--游戏控制器 -->
<script src="http://output.jsbin.com/yeqicu.js"></script>
```

在以上代码中，HTML 中有一个特殊的初始 script 元素，与其他 script 元素不同，没有 src 属性，它并不是用于链接到要加载的文件，而是直接将 JavaScript 包含在两个 script 标签之间。因为加载模块文件可能需要耗费一段时间，所以应该给玩家相应的反馈信息，告诉他们游戏正在加载中。如果将代码放在 HTML 中就可以立即运行，无需等待文件加载。

以下代码清单 16-12 显示了启动和运行游戏所需的代码。



代码清单 16-12　运行游戏
(http://jsbin.com/fociqo/edit?js,console)

```
var playerName = "Jahver";
var map = theCrypt.mapData;
game.init(map, playerName);
```

请读者隐藏控制台上的 JavaScript 面板，运行游戏，开始探索吧！游戏精彩，务必谨慎，挑战失败会损耗健康值！ 297

16.7　The Crypt——下一个目标

值得庆祝！我们已经成功创建了一个基于控制台的冒险游戏，该游戏具有模块化架构和可交换的地图数据（也就是说游戏由许多部分组成）。

回顾本书第二部分的所有内容，其核心理念就是如何有效组织程序代码，以便应对更大的程序。现在，我们掌握了私有变量、模块、模型、视图和控制器，就能够驾驭更加宏伟的项目。本书在第一部分和第二部分一直在使用 JS Bin 控制台，目的在于使读者能够心无旁骛，关注 JavaScript 语言的核心概念。在第三部分，我们将从控制台转入 HTML 和 Web 界面，成长为网络开发人员。读者会发现，在本书前两个部分中精心构建的有组织的代码将为第三部分的网络开发提供强有力的助推力。

16.8　本章小结

- 使用控制器管理模型和视图。控制器响应用户输入，更新模型，并将数据从模型传递到视图，以供显示。 298

第三部分　浏览器中的 JavaScript

本书第一部分和第二部分一直在使用 JS Bin 控制台作为编程的交互平台，目的在于让读者能够专注学习 JavaScript 代码。现在，我们已经完成上述任务，下一步将使用网页作为用户界面，将使用超文本标记语言（HTML）来指定演示内容的标题、段落和列表项目，还将使用按钮、下拉列表和文本框供用户输入。我们还会使用模板，作为从用户输入和 XMLHttpRequest 对象获取数据生成 HTML 的有效方式，为网页加载额外的数据。

本书第三部分还将介绍如何在自己的计算机上（而不是在 JS Bin 上）组织文件，并为读者在读完本书之后继续学习 JavaScript 提出一些后续建议。在本书第三部分，游戏 The Crypt 经过 HTML 改造之后，能够随着玩家探索前行的步伐每次只加载一个新位置，而不需要一次性加载所有位置。 299

第 17 章　HTML：构建网页

本章内容包括:

- 使用 HTML 显示静态内容
- 标签、元素和属性
- 常见的 HTML 元素
- 使用 JavaScript 操作网页内容
- 把游戏 The Crypt 切换到 HTML 视图

JavaScript 可以为网页添加交互性，例如对用户单击按钮做出反应、从下拉菜单中选择菜单项、更新页面的部分内容。我们已经掌握了该语言的基本知识，为下一步开发网页交互性和动态内容做好了准备。本书的重点依然是 JavaScript，但也需要学习一些 HTML，以便使用 JavaScript 来创建和操作网页的内容。用户依然可以使用 JS Bin 来添加 HTML，并在 Output 面板中查看生成的网页。在第 21 章中，读者会看到如何脱离 JS Bin 沙盒环境来组织和存放自己的 HTML、CSS 和 JavaScript 文件。

我们已经在前面章节中花了很多时间来构建一个模块化游戏 The Crypt，如果能够把该游戏从控制台上显示输出轻松切换到网页上显示输出，我们就会由衷地感觉到自己原先在构建模块化程序上所付出的努力都是值得的。现在仅仅需要几行代码就可以实现上述切换过程，简直太不可思议了!

17.1　HTML、CSS、JavaScript——构建网页

我们现在要构建一个电影评论网页：My Movie Ratings（我的电影评分），展示我们最喜欢的电影信息，以及对这些电影的评价（图 17-1）。要构建这样的页面，需要使用三种语言：

（1）HTML——我们使用 HTML 来注释页面的文本并指定要加载的媒体，使用 HTML 来指定标题、段落、列表、图像和按钮。它们是网页的基础层和基本内容，是我们希望访问者能够发现和阅读的信息。

（2）CSS——我们使用级联样式表语言（Cascading Style Sheets）指定页面的外观、颜色、字体、边距、边框等。它们是网页的表示层，是对内容的美观处理。

（3）JavaScript——我们使用 JavaScript 语言为页面添加交互性，使用 JavaScript 回应按钮单击、筛选内容、载入额外数据和弹出消息。它们是网页的行为层，正是这些用户接口小戏法让一切都变得顺利流畅，改善了用户体验，提高了使用效率。

图 17-1 包含 My Movie Ratings 页面的三个屏幕截图，并显示如何在 HTML 基础上构建 CSS 层和 JavaScript 层。



302 图 17-1 以 HTML 的内容为基础，使用 CSS 和 JavaScript 增强页面的外观样式和交互性

即使没有 CSS 和 JavaScript，用户也可以访问网页的关键信息（见图 17-1 左侧截图）。CSS 增加了视觉样式感、风格认同感和精心设计感，为访问者提供了愉悦的浏览体验。图 17-1 右侧截图显示了用户单击 Rate 按钮后 JavaScript 弹出的消息。

请读者链接到 JS Bin，去直观体验一下 My Movie Ratings 页面。基于 HTML 语言的程序在 http://jsbin.com/sayayim/edit?output，基于 CSS 和 JavaScript 的程序在 http://jsbin.com/hikuzi/edit?output（用户需要运行 JavaScript 才能使 Rate 按钮生效）。

17.1.1 加载图层

如果拥有自己的网站 jahvers.crypt，可以通过在浏览器中输入网页的网址 jahvers.crypt/movies.html 来加载 My Movie Ratings 网页。浏览器会加载由服务器使用该地址发布的 HTML 文档 movies.html。该文档中包含了 CSS 代码和 JavaScript 代码，分别在 style 和 script 标签之间：

```
<head>
    <title>My Movie Ratings</title>
    <style>
        /* 此处为 CSS 代码  */
    </style>
    <script>
        /*此处为 JavaScript 代码*/
    </script>
</head>
<body>
    <!—此处为网页显示的内容 -->
</body>
```

请读者不必担心以上这些陌生的 HTML 标签，本书将在 17.2 节中有所介绍。我们很清楚程序模块化的优势（本书第二部分全都在讲模块化），所以很乐意看到使用 HTML 文件加

载 CSS 模块文件和 JavaScript 模块文件，如下所示：

```
<head>
    <title>My Movie Ratings</title>
    <link rel="stylesheet" href="movies.css" />
    <script src="movies.js"></script>
</head>
<body>
    <!–此处为网页显示的内容-->
</body>
```

读者可以看到，script 标签位于要加载的 JavaScript 模块之前；link 标签位于要加载的 CSS 模块之前。当浏览器读取 HTML 文档并准备显示它时，将加载由 link 和 script 元素所指 303
定的 CSS 和 JavaScript 文件。在第 21 章中将介绍如何组织我们自己的 HTML、CSS 和 JavaScript 文件。现在，JS Bin 在为我们组织这些文件。

17.1.2　在 JS Bin 中加载图层

在 JS Bin 上运行程序时，可以在以下三个面板上添加代码：HTML、CSS 和 JavaScript 面板，JS Bin 会自动合并来自以上三个面板的代码，将 CSS 和 JavaScript 嵌入到 HTML 中，并在 Output 面板上显示所生成的网页。

本书不会用太多篇幅讲解 CSS，但是在示例中出现 CSS 时，读者可以自己链接到 CSS 面板去看看，其代码大多简单明了。在下一节，我们将对 HTML 进行一次简要介绍，然后在第 17.3 节重新回到 JavaScript。

17.2　HTML——简略介绍

对于电影评论网站 My Movie Ratings，我们希望在网站中包括电影标题、演员和导演列表、评分选择以及一个能提交用户决定的按钮。因此需要一种方法来分别指定哪些文本是标题、哪些文本是列表项、哪个是下拉列表、哪个是按钮。我们终于不再使用控制台了，离开了那个仅有空格和换行字符的单色世界，进入到一片奇妙多彩的 HTML 天地。

我们使用 HTML 来注释和标识嵌入到文档中的文本与媒体。注释指定了文本的每一部分在文档结构中扮演的角色；媒体可以是图像、视频、音频或其他格式的文件。

注释采用成对出现的标签形式。例如，在文档中，某部分文本有可能是标题、段落、列表项或引用，就需要有相应的各种标签。下面文本中有一个标题和一个段落，每对标签都包括一个表示开始的标签和一个表示结束的标签：

```
<h1>My Movie Ratings</h1>
<p>Brief info about my favorite movies.</p>
```

读者可以看到，HTML 使用开始和结束标签包裹该部分的文本，标题使用<h1>和</h1>标签，段落使用<p>和</p>标签，为了指定一个元素，开始和结束标签总是成对儿出现。h1 标签指定了一个标题元素，p 标签指定了一个段落元素。

当 Web 浏览器加载 HTML 文档时，会识别标签，并在内存中为页面模型创建对应的元
304 素。这些标签指定了要创建元素的类型，标签之间的文本则形成了元素的内容。

17.2.1　从空白页面开始

向页面添加内容之前，我们先来看看构成空白网页的 HTML。如果在 JS Bin 上创建一个 new bin 并查看 HTML 面板，将看到以下内容（本书已经对其格式进行了重新编排）。

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>JS Bin</title>
   </head>
   <body>
   </body>
</html>
```

第一行中的 DOCTYPE 向浏览器提供了正在使用的 HTML 版本的信息，并帮助浏览器决定如何处理和显示该页面。HTML 历经多年发展，出现了许多版本和变种。幸运的是，<!DOCTYPE html>是最新版本（目前是 HTML5）的简洁速记法。

将整个文档包裹在 html 标签中。文档包含了标题 head 和正文 body 两个部分，标题部分包含了文件的相关信息：标题和字符编码。正文部分是页面的主要内容，当用户访问网页时这些内容将显示给用户。本书中，所有网页都将使用这种基本结构。

17.2.2　添加内容

图 17-2 显示，在一个新 JS Bin 文档的 body 标签之间为 My Movie Ratings 添加一些文本，并使用适当的 HTML 标签对这些文本进行标记，分别标记为标题、副标题和段落。



305 图 17-2　标题和段落的层次结构

浏览器通过使用不同的字体大小来表示标题的层次结构：h1、h2、h3。以下代码清单 17-1 显示了用于网页 body 标签之间的 HTML。



代码清单 17-1　My Movie Ratings——标题和段落（HTML）
(http://jsbin.com/nosiwi/edit?html,output)

```
<body>        <!-- 将网页上显示的内容放在 body 标签之间 -->
   <h1>My Movie Ratings</h1>       <!-- 对页面上的主标题使用 h1 标签 -->
   <p>Brief info about my favorite movies.</p>
   <h2>Movies</h2>        <!-- 对多级副标题使用标题标签 h2，h3 等 -->
   <h3>Inside Out</h3>
   <p>An emotional adventure inside the head of a young girl.</p>
                                          <!-- 将段落包含在 p 标签中 -->
</body>
```

17.2.3　标记列表

每部电影都包括演员列表和导演列表，HTML 使用 li 标签来标记列表项元素。只有一个演员的单个列表项如下所示：

```
<li>Amy Poehler</li>
```

列表项是列表的一部分，如果我们认为列表项的顺序很重要，就使用有序列表，否则使用无序列表。对于有序列表，我们可以使用 ol 标记，对于无序列表则使用 ul 标记。图 17-3 显示了演员和导演列表，代码清单 17-2 显示了该代码。

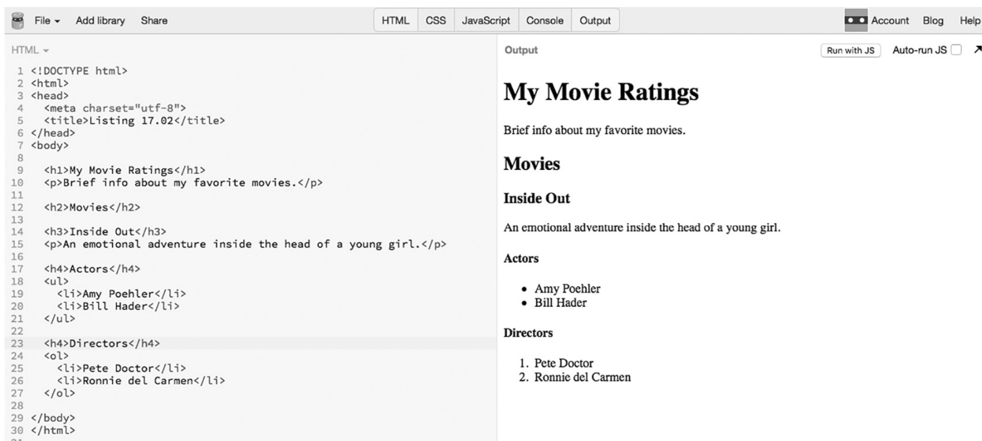


图 17-3　浏览器使用数字 1.和 2.呈现有序列表，使用项目符号 • 呈现无序列表 306



代码清单 17-2　有序和无序列表（HTML）
(http://jsbin.com/vegahe/edit?html,output)

```
<body>
   <h1>My Movie Ratings</h1>
   <p>Brief info about my favorite movies.</p>
   <h2>Movies</h2>
   <h3>Inside Out</h3>
   <p>An emotional adventure inside the head of a young girl.</p>
```

```
    <h4>Actors</h4>
    <ul>        <!-- 使用 ul 标签创建无序列表 -->
        <li>Amy Poehler</li>   <!-- 将每个列表项置于表示开始和结束的 li 标签之间 -->
        <li>Bill Hader</li>
</ul>
<h4>Directors</h4>
<ol>        <!-- 使用 ol 标签创建有序列表 -->
        <li>Pete Doctor</li>
        <li>Ronnie del Carmen</li>
    </ol>
</body>
```

如图 17-3 所示，浏览器自动为有序列表添加表示顺序的数字编号，为无序列表添加项目符号。

17.2.4 一些常见的 HTML 元素

图 17-4 是由一些常见 HTML 元素组成的网页截图。元素内容是对该元素的具体描述。

307 读者可以在 JS Bin 上访问该页面：http://jsbin.com/nuriho/edit?html,css,output。

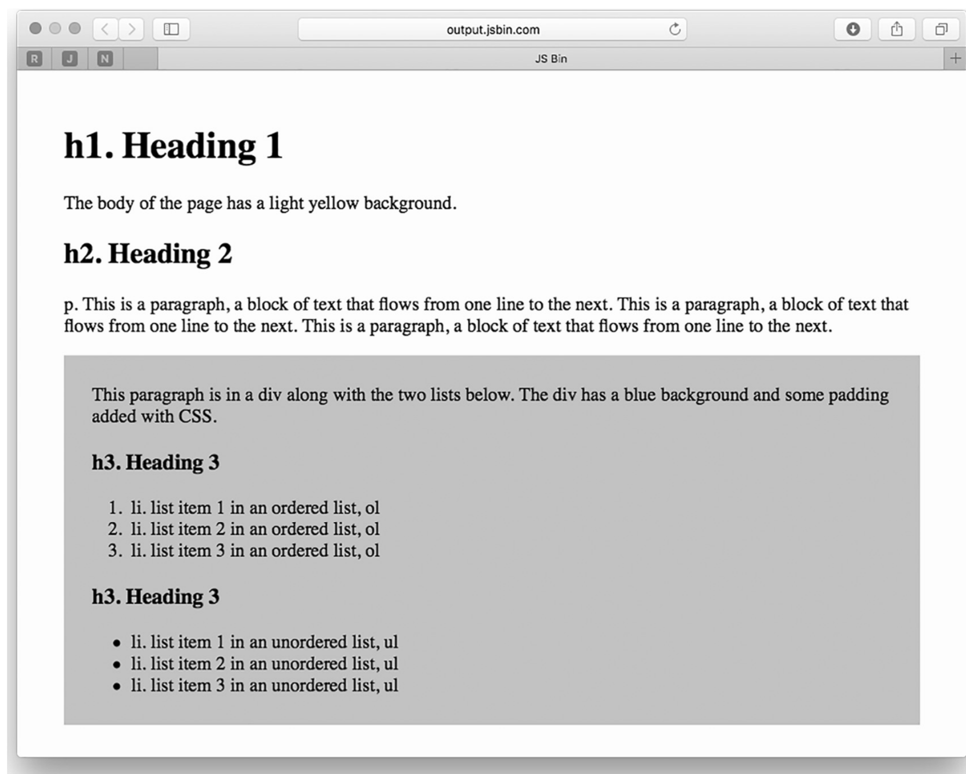


图 17-4　一个描述 HTML 元素的网页

用于创建图 17-4 中页面的标签见表 17-1。虽然 HTML 中还有很多标签，但是表中所列内容已经足够我们目前使用。

表 17-1　一些用于包裹内容的常见 HTML 元素

标　　签	元　　素	具 体 描 述
<h1>	标题	文档的主标题或某部分内容的主标题
<h2>…<h6>	副标题	按重要性降序排列的多级副标题
<p>	段落	表示段落
<div>	分隔	将本来在一起的一组元素包裹为文档的一部分
	有序列表	包裹一组对顺序有要求的列表项（例如，带有编号的列表）
	无序列表	包裹一组对顺序无要求的列表项（例如，带有项目符号的列表）
	列表项	将单个列表项包含在有序或无序列表中
<head>	头部	包裹提供文档元信息的元素，并加载文档所需的额外代码
<body>	主体	包裹页面的主要内容，即直接显示在网页上的内容

308

17.3　使用 JavaScript 向 Web 页面添加内容

为了给网站添加些乐趣，我们决定为 My Movie Ratings 网站的访问者添加随机问候语，如图 17-5 所示。用户四次访问页面，看到四条不同的问候语。

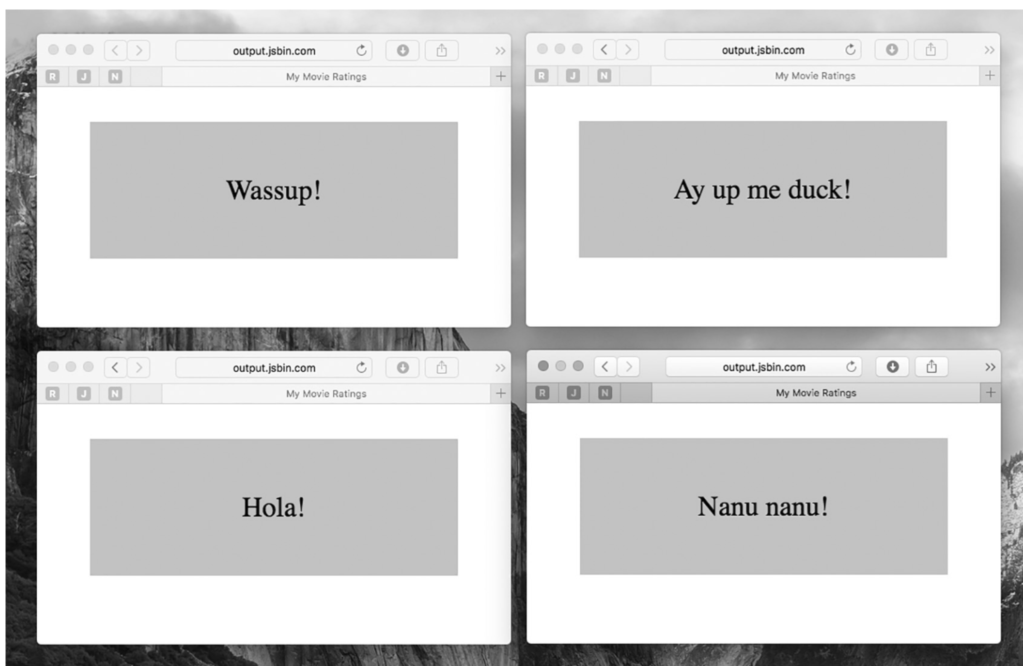


图 17-5　来自网页的随机问候语

用户每次刷新页面就会生成一条新的随机问候语。请在 JS Bin 上尝试运行，http://output.jsbin.com/mikura.html。以下两个代码清单显示如何创建该页面，运行 JavaScript，这些随机问候语就会出现。



代码清单 17-3　使用 JavaScript（HTML）向段落添加内容 (http://jsbin.com/mikura/edit?html,js,output)

```
<p id="greeting">Welcome!</p>
```



代码清单 17-4　使用 JavaScript 向段落添加内容 (http://jsbin.com/mikura/edit?html,js,output)

```
function getGreeting () {
   var hellos = [        //创建一个问候语的数组
309      "Nanu nanu!",
      "Wassup!",
      "Yo!",
      "Hello movie lover!",
      "Ay up me duck!",
      "Hola!"
   ];
   var index = Math.floor(Math.random()*hellos.length); //生成从零到小于
                                                         //数组长度的随机
                                                         //索引值
   return hellos[index];
}
function updateGreeting () {
   para.innerHTML = getGreeting();
}
var para = document.getElementById("greeting");     //获取对段落的引用
updateGreeting();        //将段落的内容设置为一条随机问候语
```

为了达到以上效果，我们需要按照下列步骤进行操作：

（1）为 HTML 中的元素分配 id。

（2）通过 id 获取 JavaScript 中元素的引用。

（3）使用该引用来更新元素的内容。

17.3.1　通过 id 获取元素

我们的目标是在段落元素中显示问候语。为了设置问候语，需要在 JavaScript 中得到对该段落的引用。首先，为 HTML 元素设置一个唯一的 id 属性：

```
<p id="greeting">Welcome!</p>
```

为了获取 JavaScript 程序中元素的引用，可以使用 document.getElementById 方法，将元素的 id 作为参数传递给该方法：

```
var para = document.getElementById("greeting");
```

网页浏览器可以让 JavaScript 代码使用 document 对象，document 对象拥有的属性和方法可让用户与页面上的元素层次结构进行交互。

程序已经通过使用 document.getElementById 获取了对段落元素的引用，在将该引用赋值给 para 变量后，就可以使用 para 变量来操作元素，然后通过设置元素的 innerHTML 属性来更新段落的内容：

```
310 para.innerHTML = "Ay up me duck!";
```

以上内容更新为：

```
<p id="greeting">Ay up me duck!</p>
```

17.3.2　函数声明

当我们需要定义一个命名函数以供后续使用时，可以使用函数声明而不是函数表达式：

```
var sayHello = function () {        //
   console.log("Hello");            //将函数表达式分配给变量
};        //
function sayHello () {              //
   console.log("Hello");            //函数声明
} //
```

在本书第 4 章，曾经提及函数声明，它是一种定义函数的替代语法。 在本书前面章节中，我们一直在使用表达式，将变量值赋值给变量，将对象赋值给变量，将数组赋值给变量，以及将函数赋值给变量，如下所示：

```
var num = 4;
var movie = {};
var actors = [];
var getRating = function () {};
```

对于命名函数，更常见的定义方法是函数声明，而不是函数表达式。从本书的第三部分开始，我们就不再使用函数的表达式，而改为使用函数声明。

17.3.3　什么？不能运行 JavaScript?

偶尔，访问者可能会使用无 JavaScript 或禁用 JavaScript 的设备来访问网站，或者他们上网的速度极慢，例如在一个酒店或火车上，需要很长时间才能加载页面上所有的 JavaScript 模块。如果页面依赖 JavaScript 来显示其内容，那么这些访问者看到的可能就是一片空白。我们考虑在页面中放一些基本内容，以免访问者浪费时间。首先确保这些底层的基本内容人人可见，而那些表层 JavaScript 所带来的额外灵动和华丽仅供有能力处理 JavaScript 的设备使用。

在以上代码清单 17-3 中，随机问候测试网页中就包含了一条初始的 HTML 问候语“Welcome!”。尽管随机问候语可以增加趣味性，但并不是必需元素。在读者学会如何向段落添加文本后，就可以一步一步地添加所有其他元素了。

17.4　显示数组中的数据

My Movie Ratings 网站上的每部电影都有一个标题和一行摘要。图 17-6 显示了网站上的一个列表，分别是三部电影： 311

My Movie Ratings

- **Inside Out**

  An emotional adventure inside the head of a young girl.

- **Tomorrowland**

  Recreating the hope and wonder of previous generations.

- **The Wizard of Oz**

  Strangers find friendship and strength on a long walk.

图 17-6　一个电影列表

给定一些电影数据，就将其插入到页面上的现有元素中。电影数据如下所示：

```
var moviesData = [
    {
        "title" : "Inside Out",
        "summary": "An emotional adventure inside the head of a young girl."
    },
    {
        "title" : "Tomorrowland",
        "summary": "Recreating the hope and wonder of previous generations."
    },
    {
        "title" : "The Wizard of Oz",
        "summary": "Strangers find friendship and strength on a long walk."
    }
];
```

我们在页面上放置了一个 id 为 movies 的 div 元素，使用 div 元素作为容器将相关元素收集到一起，如以下代码清单 17-5 所示。



代码清单 17-5　创建带有 JavaScript 的 HTML (http://jsbin.com/jakowat/edit?html,js,output)

```
<body>
    <h1>My Movie Ratings</h1>
    <div id="movies"></div>
</body>
```

每个电影都是无序列表中的列表项，浏览器会自动将列表项目添加到无序列表中。下面
312 的代码清单用于显示电影的 JavaScript。



代码清单 17-6　使用 JavaScript 构建 HTML (http://jsbin.com/jakowat/edit?html,js,output)

```
var moviesData = [ /* As above */ ];
```

```
function getMovieHTML (movie) {    //使用 getMovieHTML 函数为一个电影生成一
                                   //个 HTML 字符串
    var html = "<h3>" + movie.title + "</h3>";
    html += "<p>" + movie.summary + "</p>";
    return html;
}
function getMoviesHTML (movies) {  //使用 getMoviesHTML 函数为电影数组生成
                                   //一个 HTML 字符串
    var html = "";
    movies.forEach(function (movie) {
        html += "<li>"+getMovieHTML(movie)+"</li>";//通过在 li 标签中包裹
                        //getMovieHTML 函数的返回值来创建列表项
    });
    return "<ul>" + html + "</ul>"; //使用 ul 标签将所有列表项包裹在无序列表中
}
function render (movies) {
    var moviesDiv = document.getElementById("movies");   //使用电影的 Id
                                                         //更新元素的
                                                         //innerHTML
    moviesDiv.innerHTML = getMoviesHTML(movies);
}
render(moviesData);          //向 render 传递其所需数据，调用 render 方法
```

以上代码中，getMovieHTML 方法将 movie 对象的属性包裹在相应的开始标签和结束标签之间，构建单个影片的 HTML，如下所示：

```
var html = "<h3>" + movie.title + "</h3>";
html += "<p>" + movie.summary + "</p>";
```

getMoviesHTML 方法遍历整个电影数组，构建了所有电影的 HTML。也就是说，使用 getMovieHTML 函数可获取每个电影的 HTML，并将返回的字符串包装在 li 标签中以创建列表项元素，如下所示：

```
movies.forEach(function (movie) {
    html += "<li>" + getMovieHTML(movie) + "</li>";
});
```

然后，将生成项目列表的 HTML 放在 ul 开始标签和结束标签之间，以此获得无序列表，并返回一个完整的 HTML 字符串，如下所示：

```
return "<ul>" + html + "</ul>";
```

getMoviesHTML 函数返回的 HTML 内容（但不包含换行符和额外的空格）如下所示：

```
<ul>
    <li>
        <h3>Inside Out</h3>
        <p>An emotional adventure inside the head of a young girl.</p>
```

313

```
        </li>
        <li>
            <h3>Tomorrowland</h3>
            <p>Recreating the hope and wonder of previous generations.</p>
        </li>
        <li>
            <h3>The Wizard of Oz</h3>
            <p>Strangers find friendship and strength on a long walk.</p>
        </li>
    </ul>
```

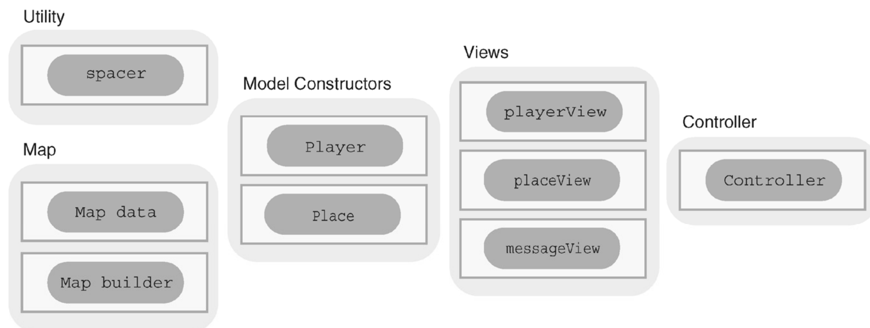
render 方法是一个渲染页面的函数，首先获取页面上目标 div 的引用，然后利用 getMoviesHTML 函数返回的 HTML 字符串设置目标 div 的 innerHTML 属性：

```
var moviesDiv = document.getElementById("movies");
moviesDiv.innerHTML = getMoviesHTML(movies);
```

为了减少使用全局变量的数量，可以将 getMovieHTML、getMoviesHTML 和 render 函数包装在立即调用函数表达式（IIFE，参见第 13 章）中，该函数表达式仅在其接口处返回 render。本章中创建的电影列表很简单，目的在于使读者集中注意力，重点学习如何生成 HTML 以及如何更新网页上的元素。

17.5 游戏 The Crypt——用网页视图方式显示玩家和地点

在前面章节中，我们已经将 The Crypt 程序分解为多个模块，每个模块都是可以单独加载的代码段，各自执行着不同的任务：数据、模型、视图和控制器（图 17-7）。



314 图 17-7 游戏 The Crypt 的各个组成模块

模块的优势在于开发者可以通过切换模块来轻松改变程序的执行动作。下面，我们就利用模块知识，按照以下步骤创建一个 HTML 版本的 The Crypt。

（1）更新玩家视图（两行代码）。

（2）更新场所视图（两行代码）。

（3）创建一个 HTML 页面，包含用来加载所有游戏模块的 script 元素，以及预备给玩家和场所视图来填充的占位符。

在所有的部件都到位之前，我们还看不到任何输出，但是又想了解正在创建的内容。图 17-8 显示了 HTML 游戏的运行情况：右侧输出显示了玩家和场所的当前状态，左侧是用户在控制台上输入的命令。



图 17-8　The Crypt 使用控制台面板输入命令和消息，在输出面板上显示更新信息

在介绍玩家和场所视图的新版本之前，我们再来体会一下模块化方法的巨大优势。render 方法可以为每个视图显示信息，为了彰显其魅力，下一节的程序中全部使用 render。

17.5.1　更新玩家和场所视图模块——render 方法

为了将游戏从控制台应用程序转变为 Web 应用程序，我们需要更新玩家和场所视图。新版本的游戏将文本插入到网页元素中来显示信息，而不是将信息输出到控制台上，这是程序所需要做出的唯一改变（图 17-9），只是一个小变化。 315

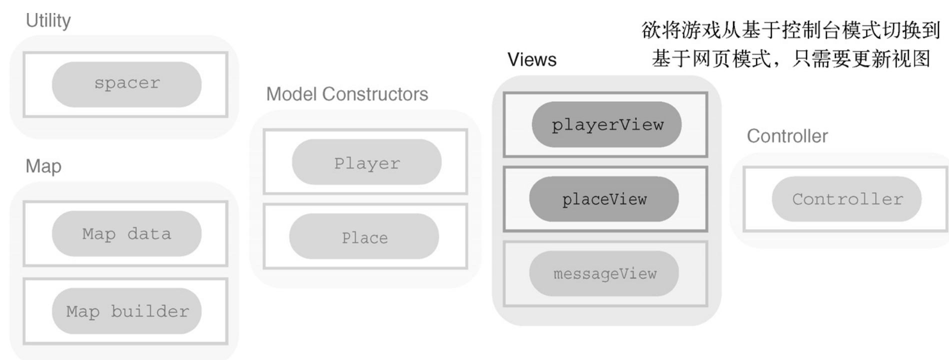


图 17-9　只需要更新视图即可将 The Crypt 游戏切换到 Web 版本

在第 15 章中，视图仅在 render 方法中使用了 console.log，正是基于这样的精心设计，现在只需要对这一个地方做出修改即可将 The Crypt 游戏切换到 Web 版本。

基于控制台的玩家视图 render 方法如下所示：

```
function render (player) {
   console.log(getInfo(player.getData()));
};
```

基于控制台的场所视图 render 方法如下所示：

```
function render (place) {
   console.log(getInfo(place.getData()));
};
```

运用本章知识更新网页元素，新的场所视图 render 方法如下所示：

```
function render (place) {
   var placeDiv = document.getElementById("place");
   placeDiv.innerHTML= getInfo(place.getData());
};
```

新的玩家视图 render 方法与上述场所视图 render 方法几乎相同：

```
function render (player) {
   var playerDiv = document.getElementById("player");
   playerDiv.innerHTML= getInfo(player.getData());
};
```

以上对 JavaScript 所做的改变满足了我们的需要。新的方法使用 document.getElementById 来获取网页上的元素引用。在创建代码清单 17-9 中的网页之前，这些元素
316 并不存在，因此无法测试新视图。请读者别着急，我们马上就可以获得这些元素。

17.5.2 更新玩家和场所视图模块——代码清单

代码清单 17-7 显示了更新 render 方法后的玩家视图，本书省略了与第 15 章中完全相同的函数体，欲查看完整的代码清单，读者可以使用代码清单中指向 JS Bin 的网址链接。对于代码清单中出现的“use strict”，将在 17.5.3 节之后进行讨论。



代码清单 17-7　基于 Web 版本的玩家视图 (http://jsbin.com/cehexi/edit?js,console)

```
(function () {
   "use strict";
   function getNameInfo (playerData) { … }
   function getHealthInfo (playerData) { … }
   function getItemsInfo (playerData) { … }
   function getTitleInfo (playerData) { … }
   function getInfo (playerData) { … }
   function render (player) {        //通过更新 render 方法来处理网页元素
      var playerDiv=document.getElementById("player");  //获取 id 为
                                                        //“player”的 div
                                                        //元素引用
      playerDiv.innerHTML=getInfo(player.getData());    //设置 div 的内容
```

```
                                                        //为 getInfo 方法生
                                                        //成的字符串
    }
    if (window.theCrypt === undefined) { window.theCrypt = {}; }
    theCrypt.playerView = { render: render };
})();
```

以下代码清单 17-8 显示了更新 render 方法后的场所视图。



代码清单 17-8　基于 Web 版本的场所视图 (http://jsbin.com/cakine/edit?js,console)

```
(function () {
    "use strict";
    function getItemsInfo (placeData) { … }
    function getExitsInfo (placeData) { … }
    function getTitleInfo (placeData) { … }
    function getInfo (placeData) { … }
    function render (place) {    //通过更新 render 方法来处理网页元素
        var placeDiv = document.getElementById("place"); //获取 id 为“place”
                                                        //的 div 元素引用
        placeDiv.innerHTML = getInfo(place.getData());//将 div 的内容设置为
                                                        //getInfo 方法生成
                                                        //的字符串
    }
    if (window.theCrypt === undefined) { window.theCrypt = {}; }
    theCrypt.placeView = { render: render };
})();
```

317

从以上代码可见，我们只需对 render 方法进行修改，就可以成功地将玩家和场所的信息显示从控制台移动到网页。虽然信息显示已经切换到 Web 版，但是目前用户还是只能在控制台上发布命令。

17.5.3　使用 JavaScript 严格模式

因为 JavaScript 已经应用在数百万个网页上，所以要确保在 JavaScript 不断发展和进化的同时，这些页面还能够继续正常运行。为了看到更多的报错提醒，为了优化语言的运作，为了给 JavaScript 未来新版本做好准备，我们可以采用严格模式来运行代码。严格模式的具体做法就是在函数的头部添加 "use strict" 表达式，如以上代码清单 17-7 和 17-8 所示。

本书中的所有代码都能够以严格模式运行，但直至代码清单 17-7 才第一次明确提出严格模式的概念，是因为本书作者考虑读者在学习 JavaScript 基础知识时就关注严格模式有可能会分散注意力。欲了解严格模式的更多信息,可以访问 www.room51.co.uk/js/strict-mode.html。

17.5.4　加载模块并在 HTML 中添加占位符

以下代码清单 17-9 显示了基于 Web 版本的完整 HTML 游戏，包括用于加载所需模块

的 script 标签。



代码清单 17-9　基于 Web 的游戏 The Crypt（HTML）(http://jsbin.com/zaxaje/edit?html,console,output)

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
   <title>The Crypt</title>   <!--更新标题 -->
</head>
<body>
   <h1>The Crypt</h1>
   <div id="place"></div>  <!-- 为场所信息包含一个 id 为“place”的 div -->
   <div id="player"></div> <!-- 为玩家信息包含一个 id 为“player”的 div -->
   <!--模块 -->       <!-- 加载游戏运行所需的所有模块 -->
   <!-- spacer -->
   <script src="http://output.jsbin.com/juneqo.js"></script>
   <!—Place 构造器-->
   <script src="http://output.jsbin.com/vuwave.js"></script>
   <!--Player 构造器-->
   <script src="http://output.jsbin.com/nonari.js"></script>
   <!--玩家视图-->
   <script src="http://output.jsbin.com/cehexi.js"></script>
   <!--场所视图-->
   <script src="http://output.jsbin.com/cakine.js"></script>
   <!--消息视图 -->
318   <script src="http://output.jsbin.com/jatofe.js"></script>
   <!-- 地图数据 -->
   <script src="http://output.jsbin.com/hozefe.js"></script>
   <!-- 地图构造器-->
   <script src="http://output.jsbin.com/paqihi.js"></script>
   <!-- 游戏控制器 -->
   <script src="http://output.jsbin.com/yeqicu.js"></script>
</body>
</html>
```

以上代码中，在 head 部分使用 title 元素来设置网页的标题。在 JS Bin 环境中工作时，页面标题并不总是可见；但是在浏览器环境中，通常使用网页标题来标记显示页面的标签页和窗口，并用标题将页面添加为书签或添加到收藏夹。以上代码中，将用于加载模块的 script 标签添加在 body 结束标签之前，这样做可以确保程序使用的两个 div 元素出现在页面上，以便以后更新玩家和场所信息。

不幸的是，由于视图中使用了 spacer 命名空间来格式化文本内容，因此这些文本内容中包含了控制台上的换行符和空格符，但是网页中根本不使用这些换行符和空格，导致在网页上所有玩家和场所信息连成一片，如图 17-10 所示。显然，这样的输出形式并不好看，但是

并不影响程序的运行。

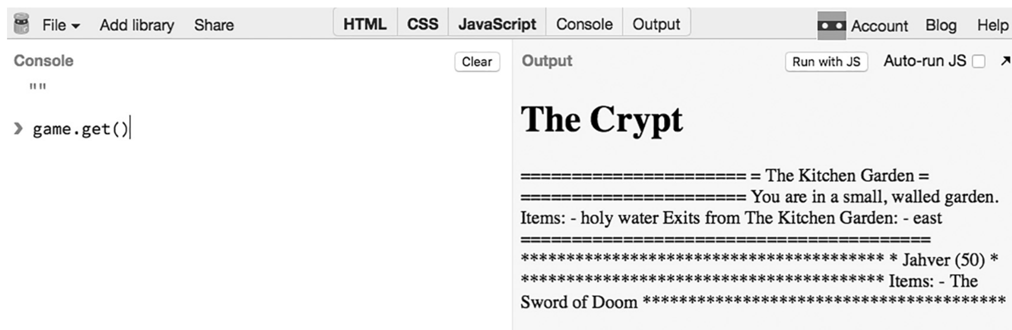


图 17-10　文本输出虽然连成一片，但是游戏依然可以运行

17.5.5　添加少许 CSS

如何解决在网页上文本输出连成一片的问题？别忘了，在 JS Bin 上还有一个 CSS 面板！ 它用于为页面上的元素指定样式，例如大小、颜色、边距、边框等。仅仅使用几行 CSS 代码，就可以让浏览器执行视图模块所生成的换行符，还可以让浏览器使用相同规格的字符，如同控制台上的字体一样。以下代码清单 17-10 就是添加到 CSS 面板上的代码。 319



代码清单 17-10　基于 Web 的游戏 The Crypt (CSS) (http://jsbin.com/toyahi/edit?css)

```
div {
    white-space: pre;
    font-family: monospace;
}
```

以上第一条规则让浏览器保留 div 元素中文本内容的空格字符（即空格和换行符）。 第二条规则指定浏览器在 div 元素中的文本使用等宽字体。页面中正是使用 div 元素来放置玩家信息和场所信息，因此现在网页上的输出格式与控制台上的输出一样整洁美观，如图 17-8 所示。

17.5.6　开始玩游戏

玩家依然通过控制台发出游戏指令 get、go 和 use，这些指令就是一些 game 方法。请打开链接 http://jsbin.com/toyahi/edit?console,output，开始玩游戏吧。玩家可以在控制台上发出以下指令：

```
>game.get()
>game.go("south")
>game.use("a rusty key", "north")
```

上述采用换行符和空格来格式化输出的方法并不理想。我们希望能使用一些合适的 HTML 标签（用于标题、段落和列表）来表明在输出中不同信息的结构。在本书第 19 章

将讨论模板，读者会看到一种更好的格式化输出方法：使用 HTML 格式化玩家信息和场所信息。

当前版本的游戏其实是一个 Web 版和控制台版的混合体：虽然使用 HTML 显示一些信息，但仍然强制玩家在控制台上输入命令。在本书第 18 章，读者将看到如何使用简单的表单元素，例如按钮、下拉列表和文本框，让玩家直接在网页上与游戏进行交互。下一节内容是为后续的完全 HTML 版本做准备，我们完全可以再次更改场所视图和玩家视图，但本书在下一节中选取了消息视图进行改造。

17.5.7 准备消息视图

为了完全以网页形式呈现用户界面，我们必须更新消息视图，以便在网页上而不是控制台上显示消息。以下代码清单 17-11 显示了新的模块代码。



代码清单 17-11 Web 版的消息视图 (http://jsbin.com/nocosej/edit?js,console)

320

```
(function () {
   "use strict";
   function getMessageInfo (message) {
      return "*** " + message + " ***";
   }
   function render (message) {
      var messageDiv = document.getElementById("messages");//在页面上显示消息
      messageDiv.innerHTML = getMessageInfo(message);
   }
   function clear () {    //定义一种方法来清除任何被显示消息
      var messageDiv = document.getElementById("messages");
      messageDiv.innerHTML = "";
   }
   if (window.theCrypt === undefined) {
      window.theCrypt = {};
   }
   theCrypt.messageView = {
      render: render,
      clear: clear        //在接口中包含 clear 方法
   };
})();
```

17.6 本章小结

- 使用 HTML 对文本进行注释，指定文本在文档结构中的作用，例如其文本是标题、列表项还是段落。
- 在 HTML 文档前放置表示其文件类型的标签：

```
<!DOCTYPE html>
```

- 使用 html 标签来包裹整个文档：

```
<html> … </html>
```

- 文档的 head 元素包含该文档的相关信息，例如文档的标题和字符编码：

```
<head>
    <meta charset="utf-8">
    <title>My Web Page</title>
</head>
```

- 文档的 body 元素包含要在网页上显示的内容：

```
<body> … </body>
```
321

- 使用与内容相应的标签，例如标题标签<h1>，<h2>，...，<h6>； 段落标签 <p>；列表项标签<li>；列表标签<ol>和<ul>。
- 将 id 属性添加到开始标签，来指定页面上的元素：

```
<p id="message"></p>
```

- 通过使用 document.getElementById 方法从 JavaScript 获取对 HTML 元素的引用：

```
var para = document.getElementById("message");
```

- 通过设置元素的 innerHTML 属性来更改元素的内容：

```
para.innerHTML = "New text for the paragraph.";
```
322

第 18 章　控件：获取用户输入

本章内容包括:

- 使用按钮启动操作
- 使用文本框和下拉列表获取用户输入
- 单击按钮时自动调用函数

在日常生活中，我们喜欢使用按钮。无论是在亚马逊上购买图书，在推特上点赞，还是在深更半夜发送醉酒后写下的电子邮件，这些形态各异的按钮都是我们难以抵挡的诱惑。下面，轮到我们来发号施令了，我们要为网页添加按钮，同时还要添加文本框和下拉列表。

在本书第 17 章中，我们已经切换到 HTML，并使用 JavaScript 向网页添加内容，但是要想获得用户输入，还是局限在控制台上。作为现代的网络应用程序，用户不必理解 JavaScript，也不必被绑在控制台上，只需通过网页就能够与程序实现交互。

本章将介绍 HTML 的 input、select 和 button 元素，让用户将信息和命令输入到文本框
323 中，从下拉列表中做出选择，然后通过单击按钮启动操作。读者将看到如何设置函数，使其在单击按钮时被自动调用。

单击按钮很有趣，下面我们就介绍如何使用按钮。

18.1　使用按钮

我们正在建立一个电影评分网站 My Movie Ratings（见第 17 章）。当访问者访问该网站时，会出现随机问候语。现在，我们希望在不重新加载整个网页的情况下，访问者也能够查看到更多的问候语。为了实现上述想法，我们决定在测试页面上添加一个按钮，用户只需单击该按钮，就可以更新一条随机问候语（图 18-1）。

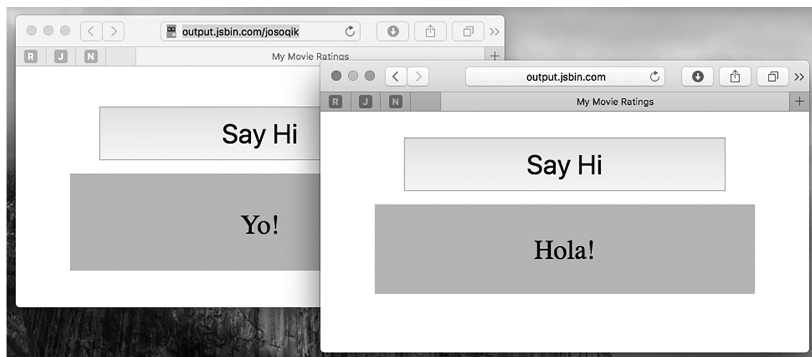


图 18-1　单击 Say Hi 按钮就会显示一条随机问候语

具体如何实现？如何使 JavaScript 程序对用户单击按钮做出响应？我们需要做三件事：

（1）向页面添加 HTML 中的一个 button 元素。

（2）编写一个函数来更新问候语。

（3）获取一个单击时就调用该函数的按钮。

18.1.1　为页面添加按钮

在 HTML 中添加按钮，只需使用 HTML 的按钮元素 button，将按钮文本放在该元素的两个标签之间即可，如图 18-2 所示。 324

```
<button>Button 1</button>
```



图 18-2　生成三个按钮的 HTML 代码及其输出

如果想在 JavaScript 中使用该按钮，需要在 HTML 中为其指定唯一的 id 属性：

```
<button id="btnGreeting">Say Hi</button>
```

以下代码清单 18-1 显示了 My Movie Ratings 测试页的关键 HTML 语句。 id 为“greeting”的段落就是我们以后插入随机问候语的地方。



代码清单 18-1　My Movie Ratings 上带按钮的问候语（HTML）(http://jsbin.com/josoqik/edit?html,output)

```
<button id="btnGreeting">Say Hi</button>
<p id="greeting">Welcome!</p>
```

输出结果如图 18-1 所示。如果读者对如何应用样式感兴趣，可以浏览 JS Bin 上的 CSS 面板。

18.1.2　编写函数来更新问候语

在第 17 章中，我们编写了两个函数 getGreeting 和 updateGreeting，用于选择一条随机的欢迎消息并更新显示。

```
function getGreeting () {
   // 返回随机问候语
};
function updateGreeting () {
   para.innerHTML = getGreeting();
};
var para = document.getElementById("greeting");
updateGreeting();
```

以上代码中，updateGreeting 函数通过设置段落元素 para 的 innerHTML 属性来更新显示。

目前，我们已经掌握了 HTML 中的按钮和段落，还有一个更新段落的函数。下面，我们希望在用户单击该按钮时能够调用 updateGreeting 函数。

18.1.3 监听单击

用户单击按钮，网页上就会弹出“Wassup！”；再次单击按钮，弹出“Hola！”……再次
325 单击按钮，弹出“Ay up me duck！”

现在，我们想发出以下指令：“亲爱的按钮，请在有人单击您时调用 updateGreeting 函数”。为了给按钮发送一条这样的指令，我们需要在 JavaScript 中引用该按钮，如下所示：

```
var btn = document.getElementById("btnGreeting");
```

在以上代码中，使用 btn 引用一个按钮，其实就是告诉按钮，当该按钮被单击时调用一个函数：

```
btn.addEventListener("click", updateGreeting);
```

addEventListener 方法命令按钮（或其调用的元素），不论何时单击按钮，只要指定的事件（event）发生就调用指定的函数。以上过程犹如 updateGreeting 函数一直等待，或监听（listening）按钮被单击的那一刻。

以下代码清单 18-2 是随机问候语测试网页的最终版本，它与第 17 章中的代码清单基本相同，但添加了两行按钮代码，按钮代码包含在立即执行的函数表达式中，以免污染全局命名空间。代码还包括一个“use strict”语句，表明浏览器使用了严格模式（见 17.5.3 节）。



代码清单 18-2 My Movie Ratings 上带按钮的问候语 (http://jsbin.com/josoqik/edit?js,output)

```
(function () {
   "use strict";
   function getGreeting () {
      var hellos=["Nanunanu!", "Wassup!", "Yo!", "Hello movie lover!",
         "Ay up me duck!", "Hola!" ];
      var index = Math.floor(Math.random() * hellos.length);
      return hellos[index];
   };
   function updateGreeting () {
      para.innerHTML = getGreeting();
   };
   var btn = document.getElementById("btnGreeting");//获取对按钮的引用
   var para = document.getElementById("greeting");
   btn.addEventListener("click",updateGreeting);    //单击按钮时调用
                                                    //updateGreeting 函数
   updateGreeting();
})();
```

请读者参看本书第 12 章，复习如何使用 Math.floor 和 Math.random 为数组生成随机索引。 326

许多奇妙事件

单击鼠标并不是函数监听的唯一事件，函数关注的事件还包括鼠标移动、页面滚动、按下键盘以及图像加载等。针对触摸屏和移动设备，敲击屏幕、在屏幕上轻划手指、强力按压屏幕以及摇动手机等也都是调用函数的事件。

目前，读者只需使用简单的单击按钮即可，如果对上述其他调用函数的奇妙事件感兴趣，请查看 Mozilla Developer Network 上的事件引用内容，网址是 https://developer.mozilla.org/en-US/docs/Web/Events。

18.2　使用 select 元素选择一个选项

My Movie Ratings 网站用于请用户对电影做出评价。用户从下拉列表中选择评分，并单击 Rate 按钮，该页面就会显示评分消息（图 18-3）。

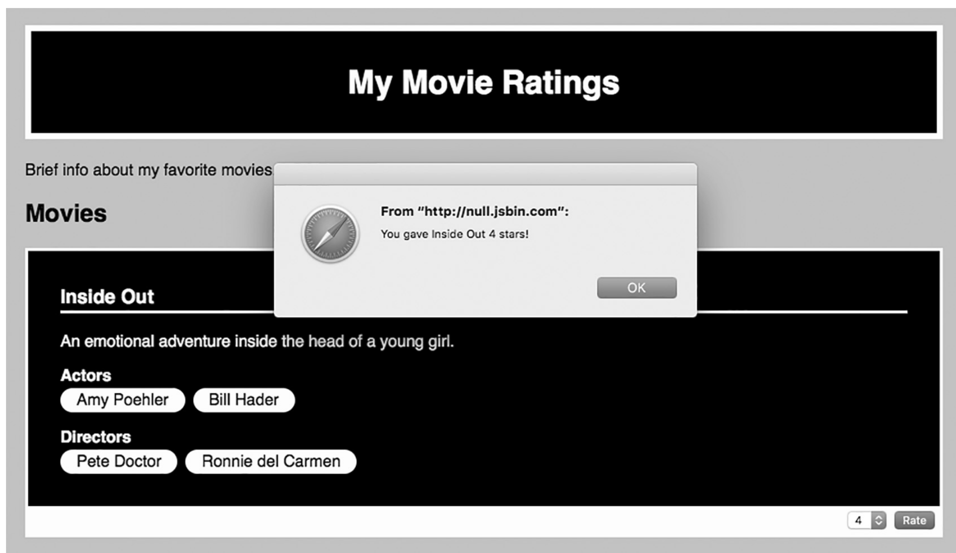


图 18-3　单击 Rate 按钮弹出带有用户评分的消息

要实现以上评分系统，我们需要做四件事：

（1）向页面中添加一个 selectHTML 元素，其 option 元素对应每个评分。

（2）向页面中添加一个 buttonHTML 元素，Rate 作为其文本。

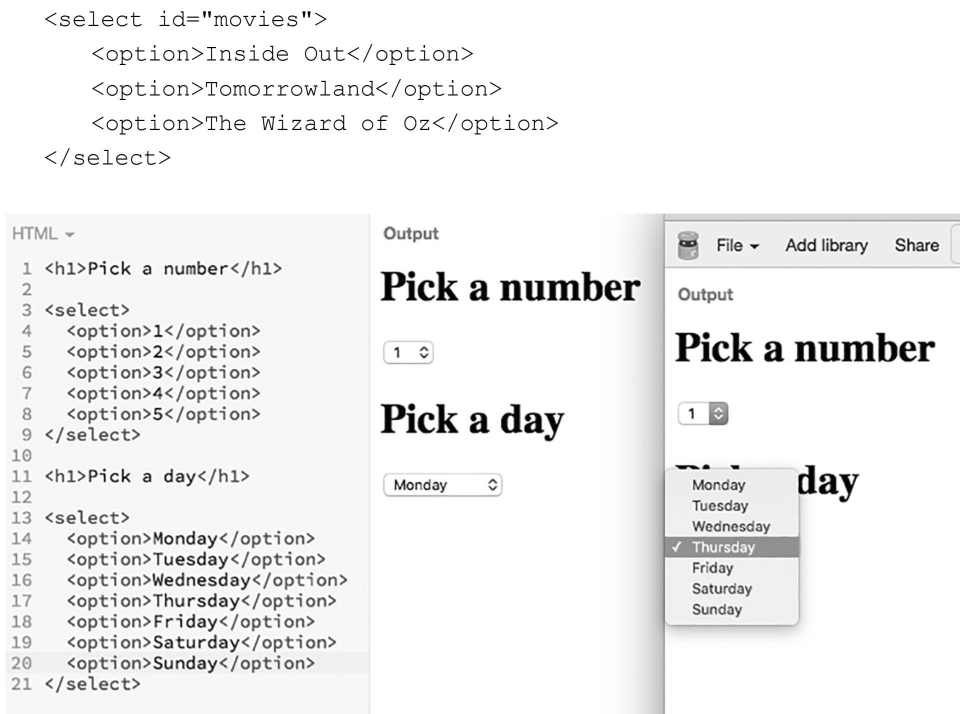
（3）编写一个函数以弹出带有评分的消息。

（4）单击按钮时调用该函数。 327

18.2.1　向页面中添加 select 元素

Web 浏览器将 select 元素呈现为下拉列表形式。我们可以使用 option 元素来指定列表中的选项（图 18-4）。

```
<select id="movies">
    <option>Inside Out</option>
    <option>Tomorrowland</option>
    <option>The Wizard of Oz</option>
</select>
```



328 图 18-4 （从左至右）两个 select 元素的 HTML 代码、其在网页上的显示以及正在进行选择的截图

我们可以在 JS Bin 上测试以上代码。首先，同时打开 HTML、Console 和 Output 三个面板，创建一个 new bin；然后，使用以上代码段中的 select 代码替换 HTML 面板的内容，就会看到有下拉列表出现在 Output 面板上；最后，在控制台提示符下输入以下命令：

```
>var dd = document.getElementById("movies")
undefined
>dd.value
Inside Out
```

从 Output 面板的下拉列表中选择不同的电影，然后使用 JavaScript 查看其新值：

```
>dd.value
The Wizard of Oz
```

我们也可以用 JavaScript 更新下拉列表的值：

```
>dd.value = "Tomorrowland"
Tomorrowland
```

在 JavaScript 中设置的值作为下拉列表框中的选项显示在 Output 面板上。

图 18-4 显示了 HTML 中的两个 select 元素，以及这两个元素在网页上的呈现形式。

以下代码清单 18-3 是 My Movie Ratings 网页主体部分的 HTML，带有一个 select 元素和一个 button，用来对电影进行评估。



代码清单 18-3　带有下拉列表的 My Movie Ratings（HTML）(http://jsbin.com/hikuzi/edit?html,output)

```
<h1>My Movie Ratings</h1>
<p>Brief info about my favorite movies.</p>
<h2>Movies</h2>
<div id="movieBox">
   <h3 id="movie">Inside Out</h3>
   <p>An emotional adventure inside the head of a young girl.</p>
   <h4>Actors</h4>
   <ul>
      <li>Amy Poehler</li>
      <li>Bill Hader</li>
   </ul>
   <h4>Directors</h4>
   <ul>
      <li>Pete Doctor</li>
      <li>Ronnie del Carmen</li>
   </ul>
   <div class="controls">
      <select id="rating">     <!-- 使用 select 元素添加一个下拉列表 -->
      <option>1</option>        <!-- 使用 option 元素将选项添加到列表 -->
      <option>2</option>
      <option selected>3</option>        <!-- 指定首选项 -->
      <option>4</option>
      <option>5</option>
      </select>
      <button id="rate">Rate</button>        <!-- 添加 button 元素 -->
   </div>
</div>
```

以上代码中，我们看到当网页上显示下拉列表时，可以添加一个 selected 属性表示用户选择了该选项（以上为第 3 个选项）。

18.2.2　对电影进行评分的函数和调用该函数的按钮

为了对影片进行评分，我们需要一个函数从下拉列表中获取用户的选择，在 select 元素中获取所选项的值，然后弹出包含该值的消息，如代码清单 18-4 所示。 329



代码清单 18-4　带有下拉列表的 My Movie Ratings (http://jsbin.com/hikuzi/edit?js,output)

```
(function () {
   "use strict";
   var rateButton=document.getElementById("rate"); //收集所使用元素的引用
   var ratingList = document.getElementById("rating");
   var movie = document.getElementById("movie");
```

```
        function rateMovie () {
            var rating = ratingList.value;    //从下拉代码清单中获取所选的选项
            var movieTitle = movie.innerHTML; //获取电影的标题
            alert("You gave " + movieTitle + " " + rating + " stars!");
                                             //使用 alert 方法弹出消息将评分存入数据库
        }
        rateButton.addEventListener("click", rateMovie);   //在单击按钮时调用
                                                           // rateMovie 方法
    })();
```

以上代码中，下拉列表的 value 属性提供了用户评分。调用浏览器提供的 alert 函数，就可以在弹出对话框中显示该评分。将代码包装在函数 rateMovie 中，使用 addEventListener 来命令 Rate 按钮，每当单击该按钮时就调用 rateMovie。

现在，用户已经可以从列表中选择评分。如果想给他们更多自由来表达意见，该怎么办？

18.3　使用文本框读取用户输入

值得庆贺，我们的电影评分网站现在已经可以向用户开放了！网站的成功让我们进一步思考：如果用户想要撰写评论，该怎么办？如何能让用户在评分时再添加一个简短的评论？
330 图 18-5 显示，该网站上已经添加了两条评论，并且正在添加第三条评论。

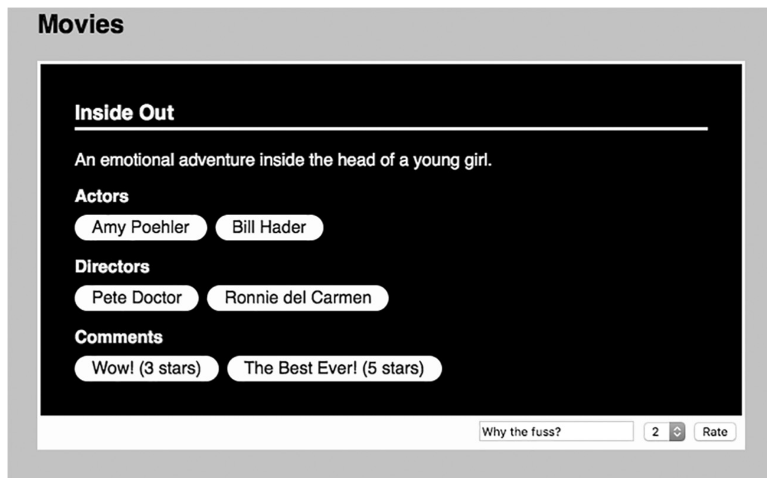


图 18-5　网页中有一个添加评论的文本框和一部分区域用来显示评论

为了让用户在网页中添加评论和评分，而不再是像上一节那样只是弹出评分消息，我们允许用户在文本框中输入评论，从下拉列表中选择评分，然后单击 Rate 按钮。为了实现上述想法，我们需要做五件事：

（1）在页面上添加一个文本框。

（2）将无序列表添加到 HTML，作为用户输入评论的地方。

（3）获取 JavaScript 中文本框的引用并访问其值。

（4）在 JavaScript 中获取对评论列表的引用。

（5）更新 rateMovie 函数，将新的评论添加到评论列表。

18.3.1　在页面中添加文本框

我们使用 HTML 中的 input 元素向页面添加文本框，其 type 属性设置为“text”，包含一个引用 JavaScript 元素的 id 属性：

```
<input type="text" id="txtComment" />
```

我们使用 input 元素在网页上显示表单控件，不包裹任何其他内容，因此没有成对儿出现的开始标签和结束标签，这称作自动闭合（self-closing）标签。 请读者注意该标签末尾的正斜杠，它表明此标签不需要结束标签。文本框处于“controls”div 中，与评分下拉列表以及 Rate 按钮一起显示，如下代码所示：

```
<div class="controls">
    <input type="text" id="txtComment" />
    <select id="rating"><!--评分选项 --></select>
    <button id="rate">Rate</button>
</div>
```

以上代码中，把 input 元素的 type 属性设置为“text”，就会在页面上显示文本框。在编程实践中，input 元素的 type 属性通常还可以是 password、submit、checkbox 和 radio。Microsoft、Apple、Google、Mozilla 和 Opera 等浏览器制造商正在致力于支持添加新的输入元素类型（例如颜色选择器、日期选择器）。想要了解有关输入元素类型的更多信息，读者可以访问 Mozilla Developer Network，网址为 https://developer.mozilla.org/en/docs/Web/HTML/Element/Input。 331

18.3.2　添加无序列表来显示评论内容

我们可以使用 ul 元素作为评论列表，并在评论内容前面加上标题：Comments。该列表具有 id 属性，以供我们在 JavaScript 中添加该列表项，如下所示：

```
<h4>Comments</h4>
<ul id="comments"></ul>
```

现在，我们在网页的 Actors（演员）列表和 Directors（导演）列表之后又成功添加了 Comments（评论）列表。

18.3.3　获取对新元素的引用

为了读取用户输入到文本框中的评论，并将列表项添加到评论列表中，我们需要在 JavaScript 代码中获取对两个元素的引用:

```
var commentsList = document.getElementById("comments"); // 列表
var commentBox = document.getElementById("txtComment"); // 文本框
```

我们可以通过 value 属性访问用户在文本框中输入的文本。要查看运行中的 value 属

性，可以在控制台上对其访问和更新。我们可以访问 My Movie Ratings 网站 http://jsbin.com/nevaxit/edit?console,output，在文本框中输入“Great Movie!”，然后在控制台提示符下输入以下命令：

```
>var txt = document.getElementById("txtComment")
undefined
>txt.value
Great Movie!
>txt.value = "Rubbish!"
Rubbish!
```

以上最后一条命令将更新网页上文本框中的内容。

18.3.4 更新 rateMovie 函数

以下代码清单 18-5 显示了该网页的全部组装部件，我们将文本框中的评论和下拉列表中的评分作为列表项目附加到现有的评论列表中。



代码清单 18-5　电影、评论和随机问候语 (http://jsbin.com/nevaxit/edit?js,output)

```
(function () {
   "use strict";
   function getById (id) {    //定义一个具有较短名称的函数，通过 id 获取元素
      return document.getElementById(id);
   }
   var rateButton = getById("rate");
   var ratingList = getById("rating");
   var movie = getById("movie");
   var commentsList = getById("comments");  //获取对元素的引用，该元素用于评论
   var commentBox = getById("txtComment");    //读取文本框中的文本
   function rateMovie () {
      var rating = ratingList.value;
      var movieTitle = movie.innerHTML;
      var comments = commentsList.innerHTML; //获取评论列表中的当前列表项
332   var comment = commentBox.value;
      comments += "<li>" + comment + " (" + rating +" stars)</li>";
                                          //使用+=运算符附加新的评论和评分
      commentsList.innerHTML = comments;      //更新评论列表
      commentBox.value = "";                  //从文本框中删除评论
   }
   rateButton.addEventListener("click",rateMovie);  //单击按钮时调用 rateMovie
   /* 随机问候语代码 - 见代码清单 18.2 */
})();
```

在以上代码清单中，需要获取对 5 个 HTML 元素的引用，因此，我们创建了一个名称较短的函数 getById，这样就不必每次都输入 document.getElementById(id)这么长的名称了。

以下代码清单 18-6 显示了更新的 HTML 版本，包括文本框、评论列表和一个随机问候语需要的 span 元素。span 用于包裹文本，我们可以描述使用该文本的目的，在本示例中为“greeting”；我们可以通过 id 来改变其样式（更改颜色、使其粗体、更改大小）；我们还可以在 JavaScript 中访问该文本（例如，添加一条随机问候语）。

body 元素的内容如下所示，其中与代码清单 18-3 相同的内容已经省略或删减，读者可以在 JS Bin 上查看完整版代码清单。



代码清单 18-6　电影、评论和随机问候语（HTML）(http://jsbin.com/nevaxit/edit?html,output)

```
<h1>My Movie Ratings</h1>
<p>Brief info about my favorite movies.
   <span id="greeting">Welcome!</span></p>    <!-- 包括一个 span，供随机问候语使用-->
<h2>Movies</h2>
<div id="movieBox">
   <h3 id="movie">Inside Out</h3>
   <p>An emotional adventure inside the head of a young girl.</p>
   <h4>Actors</h4><ul><!-- actors --></ul>
   <h4>Directors</h4><ul><!-- directors --></ul>
   <h4>Comments</h4>       <!-- 添加一个无序列表以显示评论和评分 -->
   <ul id="comments"></ul>
   <div class="controls">
      <input type="text"id="txtComment"/><!-- 添加一个文本框，供用户输入评论 -->
      <select id="rating">
         <!-- 评分选项 -->
      </select>
      <button id="rate">Rate</button>
   </div>
</div>
```

333

请读者打开链接 http://output.jsbin.com/nevaxit，运行该网页，添加一些评论和评分，并将评论和评分添加到列表中，当然，还可以顺便看到随机问候语。

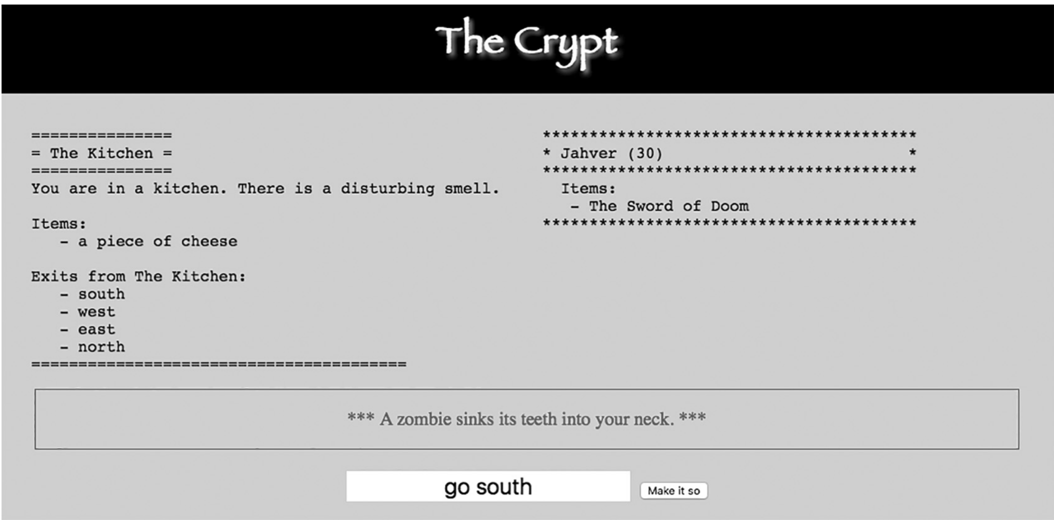
18.3.5　用 CSS 设计样式

本书第三部分中的许多示例都使用了 CSS 规则所设置的颜色、字体、边距和边框等。在本书中，并没有直接讲解与 CSS 相关的内容。如果读者对此感兴趣，可以自行阅读 CSS 规则，这些规则大多简单易懂。学习这些规则时，可以每次选取一个，还可以尝试更改某些值，删除一部分或全部规则。尽管这些规则可以使页面看上去更美观，但是如果摈弃这些规则，所有示例会运行得更加流畅。

18.4　游戏 The Crypt——玩家通过文本框发号施令

在前面章节中，游戏 The Crypt 的玩家已经能够在控制台提示符下输入 get、go 和 use 命

令。现在，我们的目标是将用户的输入从控制台转移到网页，即玩家能够在网页的文本框内输入命令，如图 18-6 所示。



334 图 18-6　玩家可以在页面底部的文本框中输入命令

为了创建如上图所示的基于网页的用户控件，我们需要完成以下三项任务：

（1）向页面添加文本框和按钮。

（2）编写一个函数将文本框中的文本转换为游戏命令。

（3）编写一个单击按钮时就会调用的函数。

我们本来可以通过更新控制器模块中的代码来包括两个新函数，但是控制器模块已经成熟稳定、运行良好，可以启动游戏并使用玩家、场所和消息的模型和视图（见第 16 章），因此不宜再做更改。较好的办法是添加一个单独的模块，专门处理输入到文本框中的命令，并调用现有的控制器来按需执行 get、go 和 use 方法。图 18-7 显示了 The Crypt 中的各个模块，其中包括新的命令（Commands）模块。



图 18-7　The Crypt 中的各模块，包括一个通过文本框执行命令的命令（Commands）模块

下面，我们先来完成以上三项任务中的第一项：在网页的 HTML 中添加文本框和按钮。

18.4.1　向页面添加控件

首先，我们需要一种方法能够让用户输入并提交命令字符串。以下代码清单 18-7 显示了 HTML 如何将文本框和按钮添加到游戏页面中。因为 player、place 和 message 的 div 元素初始为空，所以它们无法在输出 Output 面板上显示出来。



代码清单 18-7　向 The Crypt 添加控件（HTML）
(http://jsbin.com/rijage/edit?html,output)

```
<h1>The Crypt</h1>
<div id="place"></div>
<div id="player"></div>
<div id="messages"></div>
<div id="controls">
    <input type="text" id="txtCommand"> <!--添加一个 id 为 txtCommand 的文本框 -->
    <button id="btnCommand">Make it so</button><!--添加一个 id 为 btnCommand
的按钮-->
</div>
<!--包括所有模块-->
```

335

现在，文本框和按钮控件成功地出现在页面上。因为它们都拥有 id 属性，因此 JavaScript 也可以访问它们。

18.4.2　将文本框内容映射到游戏命令

表 18-1 说明了输入到文本框中的命令如何与第 16 章中创建的控制器模块中的方法匹配。

表 18-1　对比文本框命令与控制器模块中的方法

文本框命令	控制器模块中的方法
get	game.get()
go north	game.go("north")
use a rusty key north	game.use("a rusty key north")

从以上表格对比可见，用户使用文本框发号施令更加方便快捷。下一节中，我们将介绍如何把用户在文本框内输入的内容翻译成控制器能够理解的命令。

18.4.3　使用 split、join、pop 和 shift 发布命令

用户将在文本框中输入命令。我们需要将用户发出的命令转换为程序能够执行的操作，见表 18-1。为了让用户在文本框中输入的命令能够调用正确的控制器方法（get、go 或 use），首先需要用 JavaScript 对象来表示命令，该对象应该具有 type 属性，与需要调用的控制器方法匹配。表 18-2 显示了文本框命令和该命令应该生成的命令对象。

表 18-2　命令对象

文本框命令	命令对象
get	{ type: "get" }
go north	{ type: "go" direction: "north" }
use a rusty key north	{ type: "use" item: "a rusty key" direction: "north" }

在表 18-2 中，每个文本框命令的第一个单词提供了命令对象的 type。为了获得文本框命令中的每一个单独的单词，需要使用 split 将字符串转换为单词数组。

```
var commandWords = commandString.split(" ");
```

使用以上 split 方法，字符串"get a rusty key"将转换为["get","a","rusty","key"]。shift 数组方法可以删除并返回数组中的第一个元素，这正是我们所需要的抓取命令对象类型 type 的完美方法：

```
var commandWords = commandString.split(" ");
var command = {
    type: commandWords.shift();
};
```

采用以上 shift 数组方法，文本框中表示命令的单词已经从数组中删除了。["get","a", "rusty", "key"]变成为["a","rusty","key"]。

对于 go 和 use，我们可以使用 pop 数组方法来抓取 direction，即抓取 commandWords 数组的最后一个元素：

```
command.direction = commandString.pop();
```

如果 commandWords 数组中还留有任何其他元素，则将它们连接在一起形成物品的名称：

```
command.item = commandWords.join(" ");
```

例如，["a","rusty","key"]连接成"a rusty key"。

代码清单 18-8 是基于上述想法将命令字符串转换为命令对象的函数。



代码清单 18-8　将命令字符串转换为命令对象
(http://jsbin.com/repebe/edit?js,console)

```
function parseCommand (commandString) {
```

```
        var commandWords = commandString.split(" ");    //将命令字符串拆分为由
                                                        //各个单词构成的数组
        var command = {    //创建一个对象并将其分配给 command 变量
            type: commandWords.shift();       //删除数组中的第一个单词，并将其分配
                                              //给 type 属性
        };
        if(command.type === "go"||command.type==="use"){ //检查其 type 是 go 或 use
            command.direction = commandWords.pop(); //删除数组中的最后一个单词，并
                                                    //将其分配给 direction 属性
        }
        command.item = commandWords.join(" ");     //将所有剩余的单词连起来形成
                                                   //物品的名称
        return command;
    };
```

337

有命令对象在手，我们就可以轻松调用游戏控制器中的相应方法。为了有效地组织各种命令类型，可以使用 switch 结构，在多个选项之间做出决定。

18.4.4　使用 switch 在各选项间做出决定

为了让程序可以根据用户发出的不同命令采取不同的操作，我们可以使用一系列 if-else 结构，也可以使用 switch 结构，很多程序员认为 switch 结构可以让程序看起来更加整洁。我们可以使用 switch 定义一系列代码块，并根据变量或属性的值来执行其中某些代码块。以下示例中，使用 command.type 作为 switch 变量，对 switch 结构和 if-else 结构进行了简单比较：

```
switch (command.type) {
    case "get":
        game.get();
        break;
    case "go":
        game.go(command.direction);
        break;
    case "use":
        game.use(command.item,
            command.direction);
        break;
    default:
        game.renderMessage(
        "I can't do that");
}
```

```
if (command.type === "get") {
    game.get();
}
else if (command.type === "go") {
    game.go(command.direction);
}
else if (command.type === "use") {
    game.use(command.item,
    command.direction);
}
else {
    game.renderMessage("I can't do that");
}
```

在以上 switch 结构中，如果 command.type 的值为“get”，则执行第一个 case 块中的代码。如果 command.type 的值为“go”，则执行第二个 case 块中的代码。如果没有 break 语句，switch 结构将在第一个 case 块匹配后继续执行后续所有的 case 块。如果 command.type 的值与所有条件都不匹配，则执行 default 块中的代码。 338

以上两种方法其实没有本质区别。switch 结构的优势在于阅读体验更好，不足之处是额

外增加了 break 语句。其实，对 switch 结构和 if-else 结构的取舍还是取决于开发者个人喜好：如果认为 switch 结构更加整洁，那就使用 switch，只是不要忘记 break 语句。

以下代码清单 18-9 显示了 The Crypt 中的 switch 结构，该结构是 doAction 函数的一部分。

18.4.5 执行命令——监听按钮点击

如果把我们辛苦构建的游戏比作 UI 拼图，那么拼图所缺少的最后一块儿内容就是将 JavaScript 链接到 HTML。我们需要一个按钮，每当点击该按钮，就可以调用代码清单 18-9 中的 doAction 函数：

```
var commandButton = document.getElementById("btnCommand");
commandButton.addEventListener("click", doAction);
```

doAction 函数在文本框中检索文本：

```
var txtCommand = document.getElementById("txtCommand");
var commandString = txtCommand.value;
```

然后，doAction 函数解析命令字符串以创建命令对象，在 doAction 函数中使用了 switch 结构来调用匹配的控制器方法。以下代码清单 18-9 显示了命令模块的内部代码，其中用到代码清单 18-8 中的 parseCommand 函数，读者可以在 JS Bin 中查看详情。请注意，JS Bin 代码清单仅供参考，该模块无法在 JS Bin 上单独运行，如果运行该代码，会出现报错提示信息。



代码清单 18-9 命令模块
(http://jsbin.com/qedubi/edit?js,console)

```
(function () {
   "use strict";
   function parseCommand (commandString)     { /* 代码清单 18-8 */ }
   function doAction = () {    //将文本框中的文本分配给 commandString 变量
      var txtCommand = document.getElementById("txtCommand");
      var commandString = txtCommand.value;
      var command = parseCommand(commandString);//解析文本并将创建的命令对
                                                //象分配给命令变量
      theCrypt.messageView.clear();             //从消息视图中清除旧消息
      switch (command.type) {
         case "get":
            game.get();
339         break;
         case "go":
            game.go(command.direction);
            break;
         case "use":
            game.use(command.item, command.direction);
            break;
         default:
```

```
                theCrypt.messageView.render("I don't know how to do that");
            }
            txtCommand.value = ""; //清空文本框并把光标放在文本框中，为下一条命令做准备
            txtCommand.focus();
        }
        var commandButton = document.getElementById("btnCommand");    //获取对页
                                                                       //面上按钮
                                                                       //的引用
        commandButton.addEventListener("click", doAction);//每当单击按钮时，调用
                                                           //doAction 函数
    })();
```

我们总是主动为玩家着想，体现在以上代码中，在 switch 结构之后，我们将 txtCommand 的 value 属性设置为空字符串，用于清空文本框中用户的命令，其真正目的是调用文本框的 focus 方法，把光标放在文本框中，为玩家输入下一条命令做好准备。在 switch 结构之前，我们还使用消息视图的 clear 方法来清除旧消息（详见第 17 章末尾）。

在以上代码中，将命令模块包裹在立即调用函数表达式中。一旦加载网页，就会执行命令模块，并且在按钮元素上添加事件监听器。

18.4.6　进入游戏

我们并没有更改控制器模块的现有代码，而是创建了一个全新的模块：Commands 模块。该模块解析玩家的命令并调用控制器的公共方法：get、go 和 use。控制器独立于玩家接口而存在，无论是基于网页还是基于控制台，游戏的控制器模块都不需要改变，唯一改变的是控制器的接口。

图 18-8 显示了正在运行的 The Crypt 游戏，可以看到一条显示给玩家的消息。

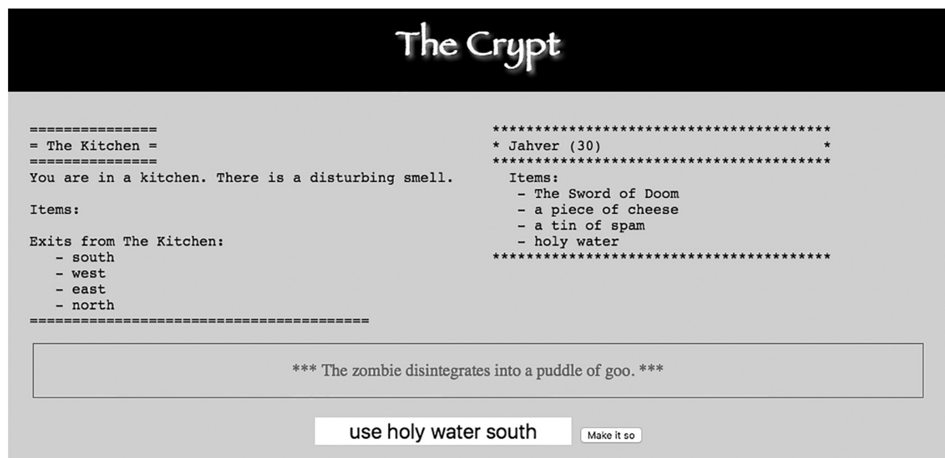


图 18-8　正在运行的 The Crypt 游戏向玩家显示一条消息

代码清单 18-7 已经是游戏最新版本的 HTML，但是我们还需要更改消息视图中的 script 元素并在新的 Commands 模块中也添加一个 script 元素： 340

```
<!-- 消息视图 -->
<script src="http://output.jsbin.com/nocosej.js"></script>
<!-- 网页控件 -->
<script src="http://output.jsbin.com/qedubi.js"></script>
```

如果读者想要玩游戏，请访问 http://output.jsbin.com/depijo；如果读者想要查看游戏的 HTML 和 JavaScript 代码，请访问 http://jsbin.com/depijo/edit?html,javascript。请务必小心僵尸！

18.5 本章小结

■ 我们想要使用按钮、下拉列表和文本框等网页控件，因此首先需要用于这些控件的 HTML 元素、元素的 id 属性以及在 JavaScript 中获取对该元素的引用；然后，就可以访问文本框或下拉列表的 value 属性，并指定一个在单击按钮时调用的函数。

■ 使用 button 元素在页面上显示按钮，把按钮上显示的文本放在开始标签和结束标签之间：

```
<button>Click Me!</button>
```

■ 在开始标签中包含 id 属性，以便从 JavaScript 代码中访问此按钮：

```
<button id="messageButton">Click Me!</button>
```

■ 在 JavaScript 中使用按钮的 id 获取对按钮的引用：

```
var btn = document.getElementById("messageButton");
```

■ 定义一个在单击按钮时可以调用的函数：

```
var showMessage = function () {
   var messageDiv = document.getElementById("message");
   messageDiv.innerHTML = "You clicked the button!";
};
```

341

■ 向按钮添加事件监听器，在单击按钮时调用函数：

```
btn.addEventListener("click", showMessage);
```

■ 使用 input 元素在页面上显示文本框，input 元素的 type 属性设置为“text”，input 元素没有结束标签：

```
<input type="text" id="userMessage" />
```

■ 使用文本框的 value 属性，获取或设置文本框中的文本：

```
var txtBox = document.getElementById("userMessage");
var message = txtBox.value;
```

■ 使用带有 option 元素的 select 元素，显示下拉列表：

```
<select>
    <option>Choice 1</option>
    <option>Choice 2</option>
    <option>Choice 3</option>
</select>
```

■ 使用 switch 结构，根据变量或属性的值执行代码：

```
switch (command.type) {
    case "go":
        // 当 command.type === "go"时，执行代码
        break;
    case "get":
        //当 command.type === "get"时，执行代码
        break;
    default:
        //都不匹配时，执行代码
}
```

342

第 19 章　模板：使用数据填充占位符

本章内容包括：

- 用一个字符串替换另一个字符串
- 使用 while 循环，重复执行代码
- 使用模板，将 HTML 与 JavaScript 分离
- 在网页中嵌入模板
- 使用 map 将一个数组转换为另一个数组

我们希望自己设计的网站不仅浏览顺畅，而且还能为用户带来乐趣。网站设计应该考虑每个页面上要包括的内容、可访问性、观感以及用户的整体体验。在建立和维护网站的工作中，开发团队的成员各有所长，会在不同阶段关注网站的不同方面。

本书一直力推使用模块化方法开发程序。我们可以看到，虽然把程序拆分为模块意味着在数量上有更多的部分需要管理，但是拆分之后，对每个部分进行单独管理会更简单、更灵活、更有针对性。在本书第二部分，所有程序都完全使用 JavaScript 语言；在第三部分，程序中出现了 JavaScript 和 HTML 交织在一起的情形，相应地出现了不受欢迎的数据流交叉，
343 有些视图将数据与带有 HTML 的 JavaScript 语言混合在一起，产生其输出（请参阅第 17 章，或最好继续读下去）：

```
var html = "<h3>" + movie.title + "</h3>";
html += "<p>" + movie.summary + "</p>";
```

本章介绍如何使用模板把 HTML 从数据和 JavaScript 中拆分出来。拆分之后，程序开发团队的设计师就可以专注于 HTML，避开棘手的 JavaScript 语法。与模块化方法同理，将 HTML 与 JavaScript 等各部分隔离开可以提高代码的灵活性、可交换性和可维护性。请不要交叉数据流！

19.1　构建一个新闻页面——重大新闻

在第 14～16 章中，我们构建了一个健身应用程序（Fitness App），用户可以使用该程序记录他们的锻炼情况，该应用程序可以在不同设备和平台上应用。现在该应用程序已经得到大众的关注，测试人员给出的报告是积极正面的，在社交媒体上该应用程序已经成为热点词汇。现在，我们打算为开发团队创建一个新闻页面，以便让开发人员、测试人员和其他感兴趣的各方人士了解我们正在进行的工作。

图 19-1 是该新闻页面的截图，页面上显示了两条新闻。开发团队的所有成员都可以给新闻

页面投稿，将投稿条目添加到中央内容管理系统（CMS）；负责管理 CMS 的工作人员将上述新闻条目作为 JavaScript 数据提供给我们；我们负责将这些数据转换为新闻页面的 HTML。

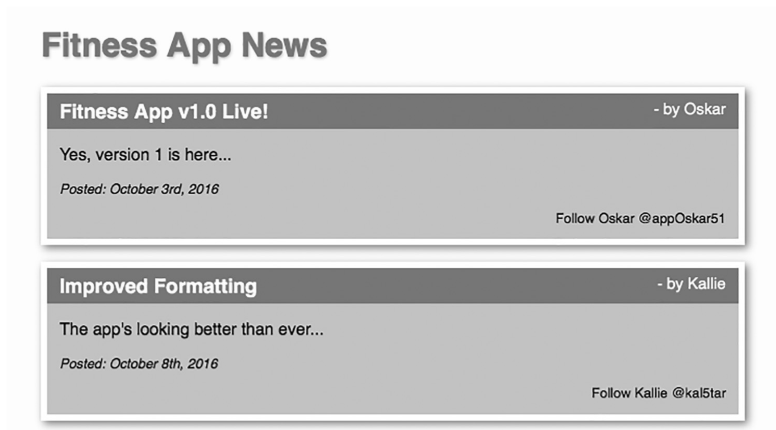


图 19-1　Fitness App News 上有两条新闻 344

19.1.1　对比新闻数据和 HTML

开发团队成员定期更新内容管理系统的新闻，该系统会为我们提供以下形式的数据：

```
var posts = [
   {
      title: "Fitness App v1.0 Live!",
      body: "Yes, version 1 is here...",
      posted: "October 3rd, 2016",
      author: "Oskar",
      social: "@appOskar51"
   },
   {
      title: "Improved Formatting",
      body: "The app's looking better than ever...",
      posted: "October 8th, 2016",
      author: "Kallie",
      social: "@kal5tar"
   }
];
```

设计人员为其中一条新闻编写了以下 HTML：

```
<div class="newsItem">
   <h3>Fitness App v1.0 Live!<span> - by Oskar</span></h3>
   <p>Yes, version 1 is here...</p>
   <p class="posted"><em>Posted: October 3rd, 2016</em></p>
   <p class="follow">Follow Oskar @appOskar51</p>
</div>
```

每一则新闻都包裹在一个 div 元素中，由一个标题和三个段落组成。标题包括一个用于新闻作者的 span 元素（我们使用 span 来为新闻作者创建不同于标题其他部分的格式）。下面，我们将介绍如何基于以上形式的数据，构造完整的 HTML 新闻条目。

19.1.2 通过连接字符串来构造 HTML

在前面章节中，我们一直在使用字符串逐步连接的方法来构造欲显示的字符串。针对一条新闻，处理方法如下所示：

```
var item = '<div class="newsItem"><h3>' + post.title;
item += '<span> - ' + post.author + '</span></h3>';
item += '<p>' + post.body + '</p>';
item += '<p class="posted"><em>Posted: ' + post.posted + '</em></p>';
item += '<p class="follow">Follow ' + post.author + ' ';
item += post.social + '</p></div>';
```

以上方法存在一个很大的缺点：将 JavaScript、数据与 HTML 三者混在一起。开发团队中有一些非常出色的设计师，他们精通 HTML，但对 JavaScript 却缺乏自信，虽然他们很乐
345 意用 HTML 构建一条新闻，但是所有的 var 和+=还有点符号让他们感觉一头雾水。即使他们会用 JavaScript，但更新代码可不像是在公园里散步那么轻松，他们眼中只有引号中的 HTML！所以，最好还是把 HTML 分离出来，让这些专家们心无旁骛，专注于他们最熟悉和擅长的 HTML。

19.1.3 使用 HTML 模板进行设计

我们请设计师为一个通用新闻模板设计出一套外观优雅、结构良好的 HTML 语句，日后我们只需向其填充最新数据即可。也就是说，请设计师为我们提供新闻条目的模板（template），如下所示：

```
<div class="newsItem">
   <h3>{{title}}<span> - {{author}}</span></h3>
   <p>{{body}}</p>
   <p class="posted"><em>Posted: {{posted}}</em></p>
   <p class="follow">Follow {{author}} {{social}}</p>
</div>
```

这样做，显然比先前那种将所有字符串逐步连接起来的方法更加方便和简洁。单引号和双引号永远不会混淆，数据的不同字段（标题、正文、作者、发布时间、社交）都清楚地标识为带有双花括号的占位符。

如果以上模板中只是使用 HTML，那么能否在网页上显示？答案是：不能。我们必须将其包裹在 script 标签中。

19.1.4 使用 script 标签包裹模板

我们使用 script 标签将 HTML 模板包裹起来，然后与网页的 HTML 其他部分一起保存。对于 script 元素，要使用非标准 type 属性，这样一来，当浏览器加载该页面时，无法识

别该 type，就会忽略该模板。script 元素的内容就不会作为输出（网页的可见）部分显示出来，也不会作为 JavaScript 参与程序的运行。

```
<script type="text/template" id="newsItemTemplate">
    <div class="newsItem">
        <h3>{{title}}<span> - {{author}}</span></h3>
        <p>{{body}}</p>
        <p class="posted"><em>Posted: {{posted}}</em></p>
        <p class="follow">Follow {{author}} {{social}}</p>
    </div>
</script>
```

如果 script 元素的 type 属性为"text / javascript"，或者 type 属性缺失，则浏览器会尝试将其内容作为 JavaScript 代码参与程序的运行。当 type 属性为"text/template"时，浏览器就会忽略其内容。

虽然浏览器在呈现页面时会忽略该模板，但我们依然可以通过其 id 属性在 JavaScript 中访问该模板：

```
var templateScript = document.getElementById("newsItemTemplate");
var templateString = templateScript.innerHTML;
```

346

以下代码清单 19-1 显示了新闻页面 body 元素中的内容：标题、放置新闻的 div 和一个包裹在 script 标签中的模板。



代码清单 19-1　Fitness App News (HTML)
(http://jsbin.com/viyuyo/edit?html,output)

```
<h1>Fitness App News</h1>
<div id="news"></news>        <!-- 包括一个 div 来放置新闻内容 -->
<script type="text/template" id="newsItemTemplate"> <!--使用一个 script 元
                                                        素来包裹新闻模板 -->
    <div class="newsItem">
        <h3>{{title}}<span> - by {{author}}</span></h3>
        <p>{{body}}</p>
        <p class="posted"><em>Posted: {{posted}}</em></p>
        <p class="follow">Follow {{author}} {{social}}</p>
    </div>
</script>
```

下一步，我们只需找到一种方法，用实际数据替换以上占位符，就可以完成该新闻内容，准备发布了。在下一节中，我们将学习如何用一个字符串替换另一个字符串，用一个占位符替换另一个占位符。

19.2　用一个字符串替换另一个字符串

用一个字符串替换另一个字符串，我们可以使用 replace 字符串方法，向其传递以下两

个参数：将欲找到并替换掉的字符串作为第一个参数，将替换后欲保留的字符串作为第二个参数，该方法就会返回一个新字符串。以下代码清单 19-2 显示了如何使用字符串"Fitness App v1.0 Live!"替换字符串"{{title}}"，在控制台上生成以下输出：

```
><h3>{{title}}</h3>
><h3>Fitness App v1.0 Live!</h3>
```



代码清单 19-2　用一个字符串替换另一个字符串
(http://jsbin.com/jeyohu/edit?js,console)

```
var before = "<h3>{{title}}</h3>";
var after = before.replace("{{title}}", "Fitness App v1.0 Live!");
                                  //查找第一个字符串并将其替换为第二个字符串
console.log(before);              //原始字符串并未更改
console.log(after);
```

replace 方法查找第一个字符串，并返回一个新的字符串：

```
before.replace(string1, string2);
```

347 以上语句用于在变量 before 字符串中搜索第一个字符串，并存储第二个字符串。

19.2.1　连续调用 replace

replace 方法对字符串进行处理，然后返回一个字符串。这意味着 replace 可以在它自己的返回值上继续调用，即 replace 可以调用自己的返回值。连续调用 replace，如下所示：

```
template
   .replace("{{title}}", "Fitness App v1.0 Live!")
   .replace("{{author}}", "Oskar");
```

假设 template 的字符串是"<h3>{{title}} - by {{author}}</h3>"，那么上述代码段的运行可以分解为以下步骤。首先，{{title}}占位符被替换：

```
"<h3>{{title}}<span> - by {{author}}</span></h3>"
.replace("{{title}}", "Fitness App v1.0 Live!")
.replace("{{author}}", "Oskar");
```

然后，{{author}}占位符被替换：

```
"<h3>Fitness App v1.0 Live!<span> - by {{author}}</span></h3>"
.replace("{{author}}", "Oskar");
```

最终结果是：

```
"<h3>Fitness App v1.0 Live!<span> - by Oskar</span></h3>";
```

代码清单 19-3 是连续两次调用 replace 的示例，在控制台上输出如下：

```
> Follow {{author}} {{social}}
> Follow Oskar @appOskar51
```



代码清单 19-3　连续调用 replace (http://jsbin.com/rebugu/edit?js,console)

```
var data = { author : "Oskar", social : "@appOskar51" };
var before = "Follow {{author}} {{social}}";
var after = before
    .replace("{{author}}",data.author)  //在原始字符串上调用 replace
    .replace("{{social}}",data.social); //对第一次调用的返回值再次调用 replace
console.log(before);
console.log(after);
```

在以上代码清单 19-3 中，将两次调用 replace 的代码写成两行的形式，目的在于让代码阅读者更容易辨识出每一次调用，对程序运行没有影响。当一些方法可以像 replace 这样，用圆点运算符再链接一个方法，我们称这些方法具有连贯接口。程序员常常设计出一整套使用连贯接口的对象和方法，使其更容易使用、阅读和理解。 348

19.3　while 循环——多次替换字符串

在上一节内容中，我们已经学会使用 replace 将一个字符串替换为另一个字符串，因此就可以编写代码来测试新闻网页。图 19-2 是一条新闻的模板，其占位符尚未替换。



图 19-2　一条新闻——显示所有占位符

每一条新闻的数据就是一个 JavaScript 对象，具有 5 个属性：title、body、author、posted 和 social:

```
var data = {
    title: "Fitness App v1.0 Live!",
    body: "Yes, version 1 is here...",
    posted: "October 3rd, 2016",
    author: "Oskar",
    social: "@appOskar51"
};
```

我们可以为每个属性调用 replace。起初，感觉一切进展顺利。图 19-3 显示了多次调用 replace 之后的新闻内容。



349 图 19-3　一条新闻——{{author}}占位符尚未替换（图的右下角）

我们距离普利策新闻奖仅仅一步之遥了！有一个占位符尚未替换，在这条新闻的右下角还有一个固执的{{author}}不肯离开。以下代码清单 19-4 是实现图 19-3 的代码。是什么地方出错了？



代码清单 19-4　为每个属性调用一次 replace
(http://jsbin.com/quroha/edit?js,html,output)

```
var data = {
    title: "Fitness App v1.0 Live!",
    body: "Yes, version 1 is here...",
    posted: "October 3rd, 2016",
    author: "Oskar",
    social: "@appOskar51"
};
var  templateScript  =  document.getElementById("newsItemTemplate");
// 获取对包裹模板的 script 元素的引用
var templateString = templateScript.innerHTML;    //提取模板的 HTML
var newsItemHTML = templateString              //使用其属性值替换每个占位符
    .replace("{{title}}", data.title)
    .replace("{{author}}", data.author)
    .replace("{{posted}}", data.posted)
    .replace("{{body}}", data.body)
    .replace("{{social}}", data.social);
var newsContainer = document.getElementById("news");  //获取对 div 的引用，
                                                      //div 将用于保存这条新闻
newsContainer.innerHTML = newsItemHTML;        //将这条新闻的 HTML 放入 div
```

我们终于发现了问题：replace 方法对于它所匹配的字符串只替换第一个，而这个新闻模板中有两个{{author}}占位符，现在只替换掉其中一个占位符。为了解决这个问题，需要针对同一个占位符两次调用 replace。

模板和 replace 代码联系太紧密了！如果设计人员打算更改模板，包含第三个{{author}}占位符（可能是为一个电子邮件链接或简短的个人介绍所准备），就必须重新更新代码，回溯到 JavaScript，针对 replace 再添加一次调用。

汲取以上示例中的教训，我们需要一种能够在有占位符时反复自动调用 replace 的代码。那么，如何以正确的次数调用 replace 呢？

19.3.1　在满足条件时重复执行代码

新闻模板中所有占位符都需要替换为相应的数据。例如，两个{{author}}都应该用 Oskar 填充，如图 19-4 的右上和右下所示。 350



图 19-4　两个 author 占位符都替换为 Oskar（右上和右下）

我们希望代码在每一次找到占位符时，都会调用 replace。伪代码如下所示：

```
while there is a placeholder {
   replace the placeholder with data
}
```

我们的目标是，发现占位符时，花括号之间的代码块就要反复执行。如果找不到占位符，可以跳过该代码块。使用 JavaScript 可以实现上述目标，如下所示：

```
while (filled.indexOf(placeholder) !== -1) {
   filled = filled.replace(placeholder, data.property);
}
```

对于以上代码，相信读者能够大致明白其功用，即使目前看不懂细节，也不必担心。下面，我们介绍 while 循环。

19.3.2　while 循环

在以上 filled 字符串中找到占位符时，while 循环重复调用 replace；在 filled 字符串中找不到占位符时，程序跳过该代码块，继续执行。通常，当某一条件为 true 时，while 循环就会重复执行一段代码，如下所示：

```
while (condition) {
   // 如果条件为真，则执行代码。
}
```

while 循环首先对条件进行评估。如果条件结果为 true，则执行代码块，这一点和 if 语句一样。while 循环与 if 语句的不同之处在于，执行代码块后，while 循环会再次评估条件。如果条件结果仍然为 true，则再次执行该代码块。也就是说，当条件为 true 时，while 循环会持续不断地执行该代码块；如果条件为 false，则跳过该代码块，程序继续执行该代码块之后的语句。 351

以下代码清单 19-5 是一个 while 循环，用于显示从 1 到 10 的整数。



代码清单 19-5　使用 while 循环进行计数
(http://jsbin.com/quroga/edit?js,console)

```
var count = 1;
```

```
while (count < 11) {
    console.log(count);
    count++;    //使用增量运算符++，将 count 的值加 1
}
```

在以上代码中，一旦 count 变量达到 11，条件值就变成为 false，因此不会将 count 的值记录到控制台。代码块条件中所使用变量的值应该不断变化，否则，如果条件的值刚开始为 true，就永远不会变成 false，该循环就会成为一个无限循环。在代码清单 19-5 中，使用了增量运算符++，为 count 的值加 1。在 JavaScript 中，有三种方法可以实现上述功能，如下所示：

```
count = count + 1;
count += 1;
count++;
```

19.3.3 替换一个可以找到的字符串

当一个占位符多次出现时，可以使用 while 循环进行多次替换。首先，需要检查我们想要替换掉的占位符是否与 indexOf 方法同时存在。如果找不到占位符字符串，indexOf 返回-1；如果能够找到该占位符，那么 indexOf 不会返回-1。while 循环的结构如下所示：

```
while (filled.indexOf(placeholder) !== -1) {
    //更改 filled
}
```

在以下代码清单 19-6 中，使用 while 循环多次替换一个字符串，直到再也找不到该字符串，在控制台上生成以下输出：

```
> Starting replacement...
> {{title}} by Oskar. Follow {{author}} {{social}}
> {{title}} by Oskar. Follow Oskar {{social}}
> ...replacement finished.
```



代码清单 19-6 使用一个带有 replace 的 while 循环 (http://jsbin.com/cabaju/edit?js,console)

352

```
var template = "{{title}} by {{author}}. Follow {{author}} {{social}}";
var filled = template;     //使用了一个新变量，以保证初始的 template 不会改变
console.log("Starting replacement...");
while (filled.indexOf("{{author}}") !== -1) {     //检查是否可以找到占位符
    filled = filled.replace("{{author}}", "Oskar");   //用一个值替换占位符
    console.log(filled);
}
console.log("...replacement finished.");
```

以上代码首先将“Starting replacement...”（“正在开始替换……”）显示在控制台上；然后，while 循环两次执行其代码块，先替换第一次出现的{{author}}，再替换第二次出现的{{author}}。因为没能找到更多的{{author}}，所以 filled.indexOf（“{{author}}”）返回值为-1，

while 循环结束。程序继续执行后续代码，将"...replacement finished."（“……替换已完成”）显示在控制台上。

19.3.4　使用正则表达式替换字符串

除了使用以上 while 循环的方法，JavaScript 还有另外一种替代字符串的方法：使用正则表达式（regular expressions）。该方法使用一种高效但比较复杂的方式来指定替换和匹配的字符。正则表达式超出了本书的范围，如果读者感兴趣，可以在本书的在线网站查询相关示例：www.room51.co.uk/js/regexp.html。

19.4　自动替换模板中的占位符

在以上健身应用程序中，显示的是用户的锻炼记录，其新闻网页显示的是一则新闻内容。在以上测验应用程序中，显示的是一组组问题和备选答案。在游戏 The Crypt 中，显示的是地图。针对以上三个应用程序，如果能够使用 HTML 模板来显示以上内容，那就非常理想了。但是，我们并不想在每次使用数据填充占位符时都重新创建一个 while 循环。是否能够让程序自动使用模板？

19.4.1　将模板占位符与对象属性相匹配

图 19-5 再次显示了一条来自健身应用程序新闻页面的新闻，尚未用数据填充占位符。



图 19-5　一条尚未填充占位符的新闻

353

将要填充占位符的数据如下所示：

```
var data = {
   title: "Fitness App v1.0 Live!",
   body: "Yes, version 1 is here...",
   posted: "October 3rd, 2016",
   author: "Oskar",
   social: "@appOskar51"
};
```

我们注意到，以上新闻中数据对象的属性名称（即“键”）与模板中占位符的名称相匹配。针对每个“键”，我们希望在程序每一次找到与其匹配的占位符时，都会使用该“键”来替换这个与其匹配的占位符。具体步骤如下：

（1）针对第一个“键”title，反复使用“Fitness App v1.0 Live!”替换{{title}}，直到没有其他{{title}}占位符可以替换。

（2）针对第二个“键”body，反复使用“Yes, version 1 is here...”替换{{body}}，直到不再有需要替换的{{body}}占位符。

（3）针对每个“键”都重复上述过程，直到所有“键”替换掉其匹配的所有占位符。

我们可以使用 Object.keys 方法来获取一个包含新闻内容属性名称的数组。

```
var keys = Object.keys(data);
```

以上述新闻为例，keys = ["title","body","posted","author","social"]。

通过使用“键”，我们可以轻轻松松地创建很多占位符。别忘了，我们可以使用方括号运算符来检索“键”的值：

```
data["title"]; // "Fitness App v1.0 Live!"
data["author"]; // "Oskar"
```

我们还可以使用一个属性的“键”来构建模板中需要匹配的占位符：

```
var placeholder = "{{" + key + "}}";
```

既然我们要使用“键”来解锁属性和占位符，对“键”的命名要尽量合理易懂（表 19-1）。

表 19-1　通过属性的键来访问属性的值并构建相应的占位符

键	值	占位符
title	data["title"]	{{title}}
body	data["body"]	{{bady}}
posted	data["posted"]	{{posted}}

354

19.4.2　为每个“键”填充所有的占位符

下面，我们把上述内容整合在一起进行编程。这些代码不仅能为健身应用程序和新闻页面模板填充数据，而且还能处理所有数据和模板的匹配问题。代码清单 19-7 显示了一个 Fill 函数，该函数基于前几节的想法，遍历一个数据对象的所有“键”并用它们的“值”替换与其匹配的占位符。



代码清单 19-7　一个用数据填充模板的函数
(http://jsbin.com/bazika/edit?js,output)

```
function fill (template, data) {
   Object.keys(data).forEach(function (key){  //遍历数据中的每个“键”
      var placeholder = "{{" + key + "}}";    //使用“键”构建占位符并检
                                              //索数据值
      var value = data[key];
      while(template.indexOf(placeholder)!==-1){//一直使用“值”替换占位符，
                                                //直到再也找不到该占位符
         template = template.replace(placeholder, value);
      }
      });
   return template;                             //返回已经被填充好的模板
```

```
}
// 测试 fill 函数
var data = {
    title: "Fitness App v1.0 Live!",
    body: "Yes, version 1 is here...",
    posted: "October 3rd, 2016",
    author: "Oskar",
    social: "@appOskar51"
};
var templateScript = document.getElementById("newsItemTemplate");
                                                //在网页上收集对模板和新闻
                                                //容器的引用
var templateString = templateScript.innerHTML;
var newsContainer = document.getElementById("news");
newsContainer.innerHTML = fill(templateString,data); //使用数据填充模板并更
                                                     //新新闻显示
```

在以上代码的最后一行，对 fill 函数进行了测试，用函数返回的 HTML 设置了 div 元素的内容。

19.4.3　使用模板构建多条新闻的列表

我们的工作初见成效。在健身应用程序的新闻页面上已经列出了一条条新闻。不足之处在于模板中的 fill 函数只能处理一条新闻。不过，也不必难过，我们知道 forEach 方法可以处理列表。因此，只要编写一个 fillList 函数就能轻松解决上述问题，如下所示。 355



代码清单 19-8　使用一个模板来构建列表 (http://jsbin.com/hilecu/edit?js,output)

```
function fill (template, data) { /* 代码清单 19-7 */ }
function fillList (template, dataArray) {
    var listString = "";
    dataArray.forEach(function (data) {//针对每条新闻都调用 fill 函数，构建列
                                       //表字符串
        listString += fill(template, data);
    });
    return listString;
}
// 测试该函数
var posts = [
    {
        title: "Fitness App v1.0 Live!",
        body: "Yes, version 1 is here...",
        posted: "October 3rd, 2016",
        author: "Oskar",
        social: "@appOskar51"
    },
```

```
{
        title: "Improved Formatting",
        body: "The app's looking better than ever...",
        posted: "October 8th, 2016",
        author: "Kallie",
        social: "@kal5tar"
    }
    /* 在 JS Bin 上还有另外两条新闻 */
];
var templateScript = document.getElementById("newsItemTemplate");
var templateString = templateScript.innerHTML;
var newsContainer = document.getElementById("news");
newsContainer.innerHTML = fillList(templateString, posts);
                                              //为每条新闻项填写模板，并
                                              //使用已完成的 HTML 字符串
                                              //更新显示
```

在以上代码的 fillList 函数中，使用了 forEach 方法来遍历数据对象数组 dataArray，将每个数据对象传递给 fill 函数，并将填充好的模板返回到 listString。

356 掌握了以上两个模板函数，现在我们热切希望得到健身应用程序新闻的独家报道权。

19.5 创建一个新闻页面——处理最新消息

下面，我们对代码进行模块化处理。开发团队的成员之间喜欢分享代码，同时他们讨厌全局污染。因此，他们希望看到一个完整的新闻页面程序，由独立数据和模板模块两部分构成，如图 19-6 所示。

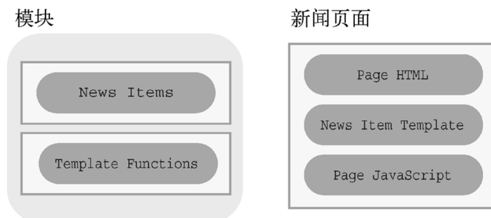


图 19-6 （左图）该页面包括两个模块：新闻内容模板和 JavaScript 代码；（右图）该页面导入这两个模块

JS Bin 上的新闻页面中包括一个包裹在 script 元素内的新闻内容模板和 JavaScript 面板上的代码，JavaScript 面板使用模块和模板来显示新闻内容。图 19-1 显示了已完成的网页，该网页上有两条新闻。

要完成上述工作，首先需要创建模块，然后将模块导入到网页中。欲知详情，请阅读以下两小节内容。

19.5.1 创建模板模块和数据模块

我们需要将模板函数放入一个模块中，将新闻数据放入另一个模块中。使用命名空间可

以避免创建太多的局部变量。

1. 模板函数

以下代码清单 19-9 显示了如何将代码清单 19-7 和 19-8 中的两个模板函数打包到一个模板模块中。



代码清单 19-9　模板模块
(http://jsbin.com/pugase/edit?js,console)

```
(function () {
   "use strict";
   function fill (template, data) {          /* 参见代码清单 19-7 */ }
   function fillList (template, dataArray) { /*参见代码清单 19-8 */ }
   if (window.gpwj === undefined) {          //确保 gpwj 在全局命名空间中
      window.gpwj = {};
   }
   gpwj.templates = {              //将两个模板函数添加到 gpwj.templates
      fill: fill,
      fillList: fillList
   };
})();
```

357

以上代码中使用了一个新的命名空间——gpwj（来自本书）。因为该模板模块不仅会应用在本书的几个程序中，而且将在许多项目中得以使用，因此非常有必要将其放入通用命名空间。需要调用这些函数时，请同时包括该命名空间：

```
var newsItemHTML = gpwj.templates.fill(newsItemTemplate, data);
```

2. 新闻数据

在现实生活中，新闻页面的新闻数据来自一个中央内容管理系统（CMS）。现在，我们通过创建一个提供虚拟数据的模块来模拟 CMS 新闻源，以后再将该模块替换为与实时 CMS 连接的模块。

以下代码清单 19-10 包括数据和一个获取数据的函数，该函数被分配给 fitnessApp.news 命名空间。



代码清单 19-10　新闻页面的数据模块
(http://jsbin.com/fupiki/edit?js,console)

```
(function () {
   "use strict";
   var posts = [    //包含一个由多条新闻构成的数组，而不是从 CMS 获取数据
      {
         title: "Fitness App v1.0 Live!",
         body: "Yes, version 1 is here...",
         posted: "October 3rd, 2016",
         author: "Oskar",
         social: "@appOskar51"
      }
```

```
        /* 在 JS Bin 中有更多的数据 */
    ];
    function getNews (numItems) {    //返回指定数目的新闻内容
        return posts.slice(0, numItems);
    }
    if (window.fitnessApp === undefined) {//确保 fitnessApp 全局命名空间存在
        window.fitnessApp = {};
    }
    fitnessApp.news = {         //将 news 命名空间添加到 fitnessApp 命名空间
        getItems: getNews
    };
})();
```

要获取一定数量的新闻，必须指定所需的条目数来调用 getItems。例如，要获取三条新闻，就使用以下代码：

358

```
var itemsData = fitnessApp.news.getItems(3);
```

如果要从 CMS 中检索一定数量的真实新闻数据，采用的代码模块接口与代码清单 19-10 中的接口相同，换句话说，要从 CMS 获取消息也需要调用 getItems 方法。正是因为使用了相同的接口，我们日后才可以轻松地将模块从现在的静态版本切换到 CMS 动态版本。

19.5.2 导入模块

为了向读者提供有关健身应用团队开发的最新要闻，我们需要创建一个简单的新闻页面，HTML 版本如下所示。



代码清单 19-11 模块化的新闻页面（HTML）(http://jsbin.com/vemufa/edit?html,output)

```
<h1>Fitness App News</h1>
<div id="news"></news>       <!--包括新闻内容的 div -->
<script type="text/template" id="newsItemTemplate">  <!--将 HTML 模板包裹
在该页面的 script 标签内-->
    <div class="newsItem">
        <h3>{{title}}<span> - by {{author}}</span></h3>
        <p>{{body}}</p>
        <p class="posted"><em>Posted: {{posted}}</em></p>
        <p class="follow">Follow {{author}} {{social}}</p>
    </div>
</script>
<!-- 模板模块 -->     <!--导入模板模块和数据模块-->
<script src="http://output.jsbin.com/pugase.js"></script>
<!-- 新闻数据 -->
<script src="http://output.jsbin.com/fupiki.js"></script>
```

为了将该程序的所有部分整合在一起，我们添加了 JavaScript，如以下代码清单 19-12 所示，先从页面检索到模板，再用新闻数据填充该模板，最后使用生成的 HTML 更新新闻 div。



代码清单 19-12　模块化的新闻页面 (http://jsbin.com/vemufa/edit?js,output)

```
var templateScript = document.getElementById("newsItemTemplate");
                                                    //从页面中获取需要的部分
var templateString = templateScript.innerHTML;
var newsContainer = document.getElementById("news");
var newsData = fitnessApp.news.getItems(3);  //使用新闻数据模块检索三条新闻
newsContainer.innerHTML=gpwj.templates.fillList(templateString, newsData);
                                  //使用模板模块生成新闻内容 HTML，并更新页面
```
359

使用模板是一种从数据生成 HTML 的常见方式。现在，我们可以从许多 JavaScript 模板库中免费得到模板，其中最流行的三个免费模板库分别是 Handlebars、Mustache 和 Pug。

以下为刚刚收到的重要新闻：因为我们整洁、模块化、可重用的编程风格深得开发团队队友们喜爱，他们誓言每天都会添加新的新闻内容。而且有些人已经把我们的 gpwj.templates 函数合并到他们自己的应用程序中了。我们也打算如法炮制，造福我们自己的游戏 The Crypt。

19.6　游戏 The Crypt——改善视图

在第 17 章中，我们为玩家和场所创建了一些基于网络的视图。这些视图显示的信息起初只是适用于在控制台上显示，因此使用了 spacer 命名空间的空格、换行符、方框和边框进行格式处理，而没有使用 HTML 格式。在本章中，我们已经学会如何使用模板分隔标记和 JavaScript 语句。下面用模板知识改进该游戏的视图，即使用适当的 HTML 标签包裹 The Crypt 游戏中的数据。

图 19-7 显示了改进之后的游戏外观。从纯文本切换到 HTML 之后，我们就可以使用 CSS 对输出进行样式处理，因此视觉效果设计将大有作为。图 19-8 显示了该项目包含的各个模块，突出显示新模块（Utility）和更新模块(Views)。

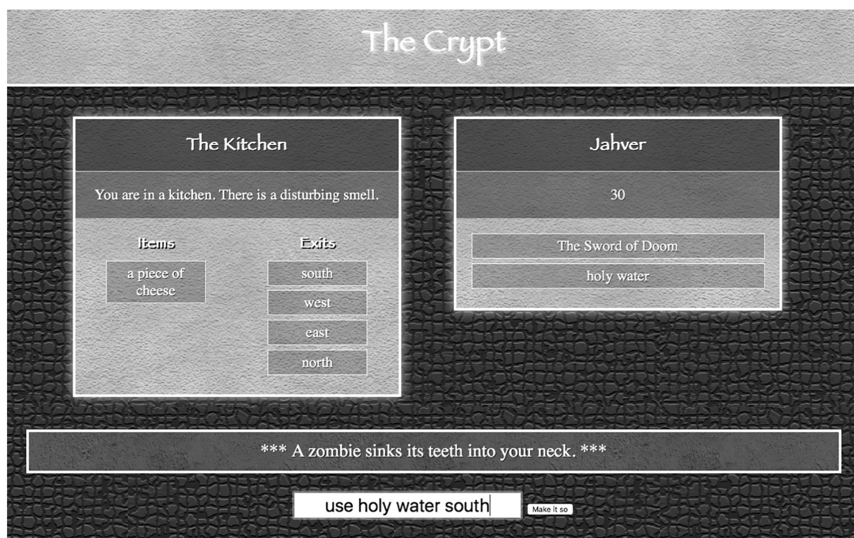


图 19-7　在游戏 The Crypt 的最新版本中使用 HTML 模板构建页面元素　360

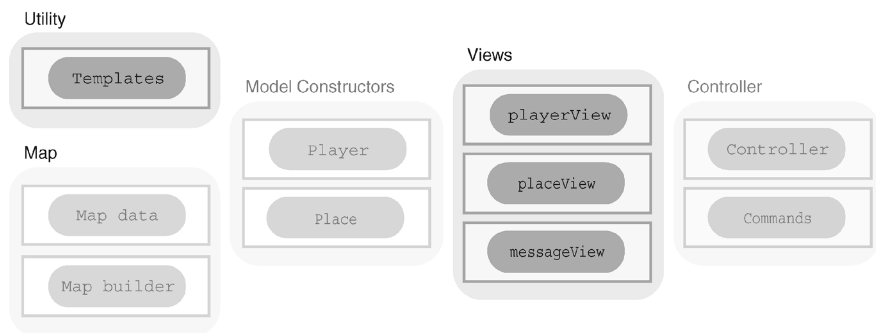


图 19-8　游戏 The Crypt 中的各个模块，突出显示新模块和更新模块

在 19.5.1 节中，我们已经创建了模板模块。下面，我们将创建更多的模板并用它们更新视图。

19.6.1　为所有视图创建 HTML 模板

以下代码清单显示，游戏中所有模板都包裹在 HTML script 标签内，这些模板是 JS Bin 完整网页的组成部分。



代码清单 19-13　带模板的 The Crypt（HTML）(http://jsbin.com/yapiyic/edit?html,output)

```
<script type="text/template" id="itemTemplate">  <!--使用 script 标签包裹页面中
                                                   的 HTML 模板 -->
   <li>{{item}}</li>
</script>
<script type="text/template" id="playerTemplate">  <!-- 给 script 标签一个非标
                                                   准类型属性，因此该标签不被执行 -->
   <h3>{{name}}</h3>
   <p>{{health}}</p>
   <ol id="playerItems"></ol>
</script>
<script type="text/template" id="placeTemplate"> <!-- 包含 id 属性，以便从
                                             JavaScript 中访问 script 元素 -->
   <h3>{{title}}</h3>
   <p>{{description}}</p>
   <div class="placePanel">
      <h4>Items:</h4>
      <ol id="placeItems"></ol>     <!-- 在模板中包含一个有序列表元素作为列表
                                    项的容器-->
   </div>
   <div class="placePanel">
      <h4>Exits:</h4>
      <ol id="placeExits"></ol>     <!-- 向列表容器提供 id，以便 JavaScript
                                    代码添加列表项-->
   </div>
```

```
</script>
<script type="text/template" id="messageTemplate">
   <p>*** {{message}} ***</p>
</script>
```

361

以上模板中并没有混入 JavaScript 代码，因此网页设计师可以使用他们熟悉的 HTML 知识更新模板。显然，以上这种针对不同的关注点分离不同模块的做法，使模板更整洁，更易于阅读，并且更易于分享。

19.6.2　针对新模板更新视图

由于游戏 The Crypt 中的模板字符串已经嵌到页面上的 HTML 中，因此我们需要更新视图模块来抓取这些模板字符串，然后使用消息、玩家和场所模型中的数据对这些模板字符串进行填充。代码清单 19-14、19-15 和 19-16 显示了新的视图代码（请注意，设计视图的目的是供控制器使用，视图本身并不能单独运行）。以上代码的 JS Bin 链接虽然不能运行代码，但是可供我们复制、克隆或更改（我们选择的）代码。

1. 消息

在该游戏的三种视图中，消息视图最简单。下面，我们就首先更新消息视图。



代码清单 19-14　使用模板的消息视图 (http://jsbin.com/jojeyo/edit?js,console)

```
(function () {
   "use strict";
   var messageDiv = document.getElementById("messages");  //收集对网页上
                                                          //元素的引用
   var templateScript = document.getElementById("messageTemplate");
   var template = templateScript.innerHTML;
   function render (message) {     //定义一个函数以填充消息模板并更新显示
      var data = { message: message };
      messageDiv.innerHTML = gpwj.templates.fill(template, data);
   }
   function clear () {             //定义一个函数清空消息 div，清除显示
      messageDiv.innerHTML = "";
   }
   if (window.theCrypt === undefined) {
      window.theCrypt = {};
   }
   theCrypt.messageView = {        //将这两个函数添加到 messageView 命名空间
      render: render,
      clear: clear
   };
})();
```

在以上消息视图中，传递给 render 方法的消息只是字符串，但是 template.fill 方法面向对象（该方法使用对象的“键”创建占位符和查找“值”），因此 render 会从消息中创建一个

362 对象 data 作为参数，消息"hello"变成{message："hello"}，然后，fill 方法将使用“值”"hello"替换{{message}}占位符。

2. 玩家

因为每个玩家都拥有多个物品，所以玩家视图比消息视图稍微复杂。除了有一个模板字符串显示玩家的名字和健康值，玩家视图还需要另一个字符串来显示物品。

下一个代码清单显示新的玩家视图，与物品相关的代码以粗体显示。



代码清单 19-15　使用模板的玩家视图
(http://jsbin.com/suyona/edit?js,console)

```
(function () {
"use strict";
var playerDiv = document.getElementById("player");
var playerScript = document.getElementById("playerTemplate");
var itemScript = document.getElementById("itemTemplate");
var playerTemplate = playerScript.innerHTML;
var itemTemplate = itemScript.innerHTML;
function render (player) {
    var data = player.getData();
    var itemsDiv;
    var items = data.items.map(function (itemName){  //使用 map 从物品名称
                                                      //字符串数组创建物品
                                                      //对象数组
        return { item : itemName };
    });
    playerDiv.innerHTML = gpwj.templates.fill(playerTemplate, data);
                                          //将从玩家模板生成的 HTML 添加到页面
    itemsDiv = document.getElementById("playerItems"); //playerItems div
                                                      //刚刚添加到页面，获
                                                      //取对它的引用
    itemsDiv.innerHTML = gpwj.templates.fillList(itemTemplate, items);
                                          //将物品的 HTML 添加到 playerItems div
    }
    if (window.theCrypt === undefined) {
        window.theCrypt = {};
    }
    theCrypt.playerView = {
        render: render
    };
})();
```

在以上代码中，玩家的物品数据是一个字符串数组。因为 fillList 需要一个对象数组，所以使用 map 将数组内容从字符串转换为对象。传递一个函数到 map 方法，map 使用该函数创建的新元素构建了一个新的数组，也就是说，数组["a lamp", "a key"]变成为[{ item : "a
363 lamp" }, { item : "a key" }]。

在添加玩家的 HTML 之前，由 fillList 生成的物品 HTML 是无法添加到页面上的，因为玩家 HTML 中包含 ol 元素，该元素将用于盛放物品。

```
//将填充好的玩家模板添加到页面
playerDiv.innerHTML = gpwj.templates.fill(playerTemplate, data);
//playerItems ol 刚刚添加到页面，获取对它的引用
itemsDiv = document.getElementById("playerItems");
//将物品列表的 HTML 添加到 playerItems ol 元素
itemsDiv.innerHTML = gpwj.templates.fillList(itemTemplate, items);
```

3. 场所

场所视图是以上三种视图中最复杂的一个，因为每一个场所都包含物品和出口。不过，显示出口的方法与显示物品的方法相同，因此，所谓的复杂只是增加了一次重复。如同物品数据一样，出口数据也是一个字符串数组，因此二者可以使用同一个模板。

以下代码是对场所视图的更新，其中出口代码以粗体显示。



代码清单 19-16　使用模板的场所视图 (http://jsbin.com/yoquna/edit?js,console)

```
(function () {
   "use strict";
   var placeDiv = document.getElementById("place");
   var placeScript = document.getElementById("placeTemplate");
   var itemScript = document.getElementById("itemTemplate");
   var placeTemplate = placeScript.innerHTML;
   var itemTemplate = itemScript.innerHTML;
   function render (place) {
      var data = place.getData();
      var itemsDiv;
      var exitsDiv;
      var items = data.items.map(function (itemName) {
         return { item : itemName };
      });
      var exits = data.exits.map(function (exitName){ //使用 map 从字符
                                                       //串数组获取对象
                                                       //数组
         return { item : exitName };
      });
      placeDiv.innerHTML = gpwj.templates.fill(placeTemplate, data);
      itemsDiv = document.getElementById("placeItems");
      itemsDiv.innerHTML    =    gpwj.templates.fillList(itemTemplate,
items);
      exitsDiv = document.getElementById("placeExits");//一旦 placeExitsol
        //被添加，就可以获得对 placeExits ol 的引用，并用出口的 HTML 来填充它
      exitsDiv.innerHTML = gpwj.templates.fillList(itemTemplate, exits);
   }
```

364

```
        if (window.theCrypt === undefined) {
            window.theCrypt = {};
        }
        theCrypt.placeView = {
            render: render
        };
    })();
```

以上代码中，再次使用 map 将物品和出口数据中的字符串数组转换为对象数组。

19.6.3　进入游戏 The Crypt

在 19.6.1 节中出现的代码清单 19-13 是 The Crypt 在 JS Bin 上的运行示例，该 HTML 代码中使用了模板和三个新视图，导入这些新模块的 script 如下所示：

```
<!-- gpwj.模板 -->
<script src="http://output.jsbin.com/pugase.js"></script>
<!--玩家视图 -->
<script src="http://output.jsbin.com/suyona.js"></script>
<!--场所视图-->
<script src="http://output.jsbin.com/yoquna.js"></script>
<!--消息视图-->
<script src="http://output.jsbin.com/jojeyo.js"></script>
```

请单击 http://output.jsbin.com/yapiyic，开始玩游戏吧！

19.7　本章小结

■ 使用 replace 字符串方法，将一个字符串替换为另一个字符串：

```
"One too three".replace("too", "two");        //返回"One two three"
```

■ 反复调用 replace，替换多个字符串：

```
"One too three".replace("One", "Far").replace("three", "long");
                                              //返回"Far too long"
```

■ 在条件为 true 时，使用 while 循环执行代码块：

```
var count = 10;
while (count > 5) {      //每一行显示一个数字，分 5 行显示 10 9 8 7 6
    console.log(count);
    count = count - 1;
}
```

■ 使用 map 数组方法根据现有数组的元素创建一个新数组。作为参数传递给 map 的函数会基于旧值返回一个新值：

365

```
var planets = ["Mercury", "Venus"];
```

```
var bigPlanets = planets.map(function (oldValue) {
    return oldValue + " becomes " + oldValue.toUpperCase();
});
//bigPlanets === ["Mercury becomes MERCURY", "Venus becomes VENUS"]
```

- 使用带有占位符的模板字符串，以免将 JavaScript 与显示字符串混淆：

```
var templateString = "<h3>{{title}}</h3><p>{{body}}</p>";
```

- 使用具有非标准类型属性的 script 标签将模板字符串嵌入 HTML：

```
<script type="text/template" id="postTemplate">
    <h3>{{title}}</h3><p>{{body}}</p>
</script>
```

- 通过 script 标签的 id 属性和元素的 innerHTML 属性，可以在 JavaScript 中访问模板：

```
var templateString = document.getElementById("postTemplate").innerHTML;
```

- 使用对象中的数据填充模板，返回一个用对象的属性替换占位符后的新字符串：

```
var data = {title: "Out of Office", body: "I'm going on an adventure!"};
var template = "<h3>{{title}}</h3><p>{{body}}</p>";
gpwj.templates.fill(template, data);
//返回 "<h3>Out of Office</h3><p>I'm going on an adventure!</p>"
```

366

第 20 章　XHR：加载数据

本章内容包括：

- 使用 XMLHttpRequest 对象加载数据
- 加载数据后调用函数
- 更新带有加载数据的视图
- JavaScript 对象表示法（JSON）
- 将 JSON 文本转换为 JavaScript 对象

在日历、电影数据库和冒险游戏中都会使用大量的数据，新闻网站也会有实时更新的重要消息和体育赛事比分。有时候，网页用户并不希望一次性加载所有数据，也不希望依靠不断刷新网页才能获取最新信息。如果页面在加载之后，还能够访问实时更新的数据，那就太棒了！这样一来，我们不需要重新加载一个完整的页面，就可以独立更新股票价格、推特、评论、赛事比分以及僵尸的健康值。

本章将介绍如何在互联网上伸出双臂，在应用程序运行的过程中抓取数据。例如，当我们在健身应用程序的各个用户之间切换时，可以加载不同用户的锻炼数据；当玩家在解决游戏 The Crypt 难题时，可以一个坟墓一个坟墓地加载各个位置的数据。

367

20.1　构建一个健身应用——检索用户数据

我们一直在构建一个健身应用程序，让用户跟踪记录他们的运动时间（见第 14～16 章）。该应用程序可以将 JavaScript 数据转换为用户模型，选用一种视图在控制台上显示用户信息，并在控制台提示符处接受用户的输入。在这个开发项目中，我们承担的具体任务如下：

（1）以字符串形式检索用户数据。

（2）将用户数据转换为用户模型。

（3）显示用户数据。

（4）为用户添加会话提供一个接口。

在前面章节中，我们已经完成任务（2）～（4），下面着手完成任务（1）：通过互联网检索用户数据。我们希望在使用该应用程序时能够在不同用户之间进行切换，如图 20-1 所示。

图 20-1 中显示了一个 app.loadUser 方法，用于为第二个用户和第三个用户加载数据。在得到所需数据之前，首先需要某种方法为每个用户指定数据的位置。

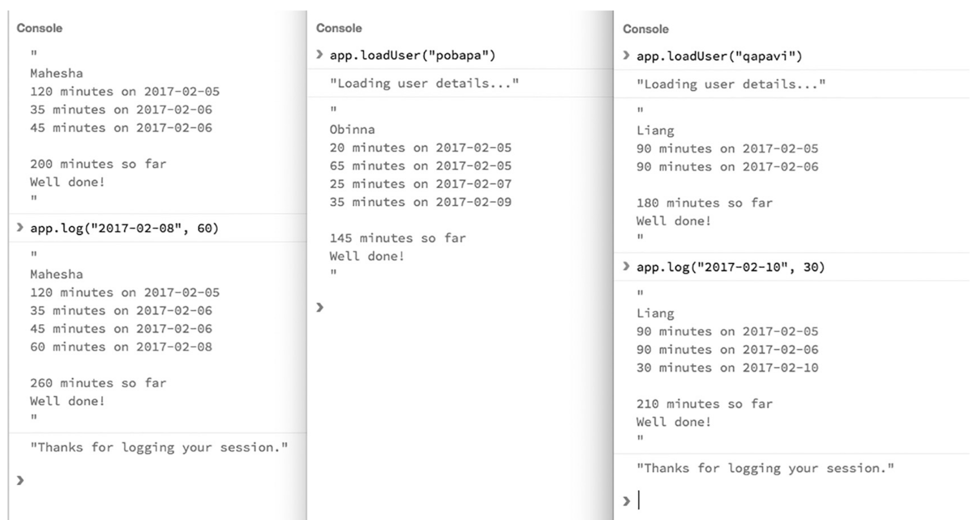


图 20-1　使用健身应用程序时在不同用户之间进行切换

20.1.1　查找用户数据

用户正在不同的平台和不同的设备上使用着不同版本的健身应用程序，但是所有版本都会使用相同的数据，该数据由健身应用中央服务器提供，如图 20-2 所示。368

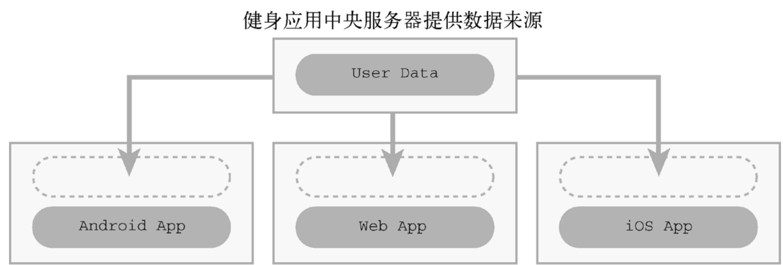


图 20-2　该应用程序的所有版本都使用相同的数据来源

在前面章节中，我们对健身应用程序的开发是针对单个用户的，因此使用的数据是静态文件。现在，我们希望在不用重新加载整个应用程序，也无需加载所有用户数据的情况下切换用户（我们希望拥有大量用户！）。我们需要定义一个 loadUser 方法，针对某个用户 ID，加载该用户的数据，并在控制台上显示：

```
>app.loadUser("qiwizo")
Loading user details...
Mahesha
120 minutes on 2017-02-05
35 minutes on 2017-02-06
45 minutes on 2017-02-06
200 minutes so far
Well done!
```

用户 ID 与 JS Bin 上的文件相对应，在实际应用中，这些 ID 可能来自数据库中的 ID，也可能是一些不重复的用户名。每个文件仅包含一个用户的数据，以下是在 http://output.jsbin.com/qiwizo.json 上 Mahesha 的数据：

```
{
    "name" : "Mahesha",
    "sessions" : [
        {"sessionDate": "2017-02-05", "duration": 120},
        {"sessionDate": "2017-02-06", "duration": 35},
        {"sessionDate": "2017-02-06", "duration": 45}
    ]
}
```

在运行程序时，如何为一个玩家加载数据？秘密在于一个命名古怪的 XMLHttpRequest
369 对象。在使用这个秘密武器之前，读者有必要先了解一下如何使用远程数据。

20.1.2 加载用户数据——概要

我们需要为这个健身应用程序加载用户数据。每当用户调用 loadUser 方法时，应用程序就会通过 Internet 访问查找数据、检索数据然后使用数据。为了实现以上步骤，健身应用程序需要：

- 用户 ID
- 用于查找数据的 URL
- 检索数据后所调用的函数

对于以上第三项需求，我们还需要进一步考虑。因为数据是通过互联网进行传输的，所以根本不知道这些数据是存放在世界各地的哪一台计算机上。因此，找到和检索数据都需要花费时间（希望是毫秒，但也许是秒）。开发程序需要使用数据才能创建新用户，并更新显示。我们定义了一个在加载数据时会调用的函数：回调函数（callback function）。其实，读者在第 18 章曾经遇到过回调函数：使用按钮时，需要一个在单击按钮时就会调用的函数，仿佛该函数正在监听单击按钮事件。为了加载用户数据，回调函数一直在监听即将发生的加载事件。加载用户数据并使用该数据更新应用程序的代码，如下所示：

```
function updateUser (userData) {
    // 获取数据，创建一个新用户
    // 使用视图更新显示
}
function loadData (id, callback) {
    // 使用 id 构建 URL
    // 数据加载时，让程序运行 callback 函数
    // 让程序从 URL 获取数据
}
loadData("qiwizo", updateUser);
```

以上代码中，向 loadData 函数传递了一个用户 id（qiwizo）和一个 updateUser 函数，在数据加载时调用 updateUser。

以上就是检索数据所涉及的大致流程，下面将介绍实施细节。

20.1.3　加载用户数据——XMLHttpRequest 构造函数

如你所见，XMLHttpRequest 构造函数的名称中只有 Request（请求）还算合理，整体来看这个名称有些怪异。现在，我们想要向某台计算机请求信息，该计算机有可能是世界上任何地方的一台服务器。读者可以将 XML 和 HTTP 作为家庭作业继续研究。从现在开始，本书将 XMLHttpRequest 简称为 XHR 构造函数，这个名字不仅短小精干，而且时髦可爱。但是，在代码中依然使用全称 XMLHttpRequest。 370

浏览器为我们提供 XHR 构造函数，我们可以使用它来创建 XHR 对象，该对象中包括从互联网请求资源的方法。我们将 URL 和待调用函数传递给 XHR 对象，当加载数据时，就会调用该函数。图 20-3 显示了在健身应用程序中，使用 XHR 对象为用户加载数据时检索到的数据。

```
Console
  "{
    \"name\" : \"Mahesha\",
    \"sessions\" : [
      {\"sessionDate\": \"2017-02-05\", \"duration\": 120},
      {\"sessionDate\": \"2017-02-06\", \"duration\": 35},
      {\"sessionDate\": \"2017-02-06\", \"duration\": 45}
    ]
  }"
```

图 20-3　XHR 对象返回的数据字符串

以下代码清单 20-1 显示，请求数据需要 5 个步骤：

（1）使用 XHR 构造函数创建 XHR 对象。

（2）声明或构建 URL。

（3）为 XHR 对象提供在加载数据时就会调用的函数。

（4）调用 open 方法，将其传递给 URL。

（5）调用 send 方法，以启动对数据的请求。



代码清单 20-1　使用 XHR 对象加载用户数据 (http://jsbin.com/qakofo/edit?js,console)

```
var xhr = new XMLHttpRequest();  //步骤(1)使用 XHR 构造函数创建一个新的 XHR 对象
var url = "http://output.jsbin.com/qiwizo.json";   //步骤(2)存储一个用于
                                                    //查找数据的 URL
xhr.addEventListener("load", function () {    //步骤(3)命令 XHR 对象在数据
                                               //加载时就调用函数
   console.log(xhr.responseText);
});
xhr.open("GET", url);                 //步骤(4)调用 open，确定 URL
xhr.send();                           //步骤(5)调用 send，启动对数据的请求
```

在以上代码中，首先，使用 new 关键字调用 XHR 构造函数，XHR 构造函数会创建并返

回一个 XHR 对象；然后，调用对象的方法可以设置和启动请求；再提供一个函数，一旦数据加载后就调用该函数。在代码清单 20-1 中，使用了函数表达式作为 addEventListener 的第二个参数，其实也可以使用先前定义过的函数名称，一旦数据加载后，数据将自动分配给 XHR 对象的 responseText 属性，回调函数可以通过该属性访问数据。调用 open 函数的第一个参数“GET”是一个表示请求的 HTTP 方法或动词。在本章中，因为一直在介绍如何获取
371 数据，所以一直在使用 GET 动词；实际上，还有一些其他动词，例如 POST、PUT 和 DELETE。

在上图 20-3 中，可以看到数据以文本形式返回。文本显示在双引号之间。数据的格式采用 JavaScript 对象表示法（JSON），将在 20.2 节中详细介绍。对于文本和字符串，不能像 JavaScript 对象一样使用 data.name 或 data ["sessions"]等表达式来访问数据。

幸运的是，有一种简单的方法将 JSON 文本转换为 JavaScript 对象。

20.1.4 加载用户数据——使用 JSON.parse 解析 XHR 响应

为健身应用用户加载数据后，XHR 请求运行回调函数。数据作为字符串自动分配给 XHR 对象的 responseText 属性。为了将字符串转换为 JavaScript 对象的属性，并确保可以访问该属性，例如 name 和 sessions，可以使用 JavaScript 的 JSON.parse 方法。

```
var dataAsObject = JSON.parse(dataAsString);
```

用户数据得以解析并记录到控制台上，如图 20-4 显示。现在，用户数据不仅是字符串，而且是具有属性的对象（请比较图 20-3 和图 20-4）。

```
Console
[object Object] {
  name: "Mahesha",
  sessions: [[object Object] {
  duration: 120,
  sessionDate: "2017-02-05"
}, [object Object] {
  duration: 35,
  sessionDate: "2017-02-06"
}, [object Object] {
  duration: 45,
  sessionDate: "2017-02-06"
}]
}
```

图 20-4 一旦 JSON 文本得以解析，就得到一个标准的 JavaScript 对象

以下代码清单 20-2 是对代码清单 20-1 的更新，在输出数据之前，先对加载文本进行解
372 析，目的是要获取一个 JavaScript 对象。



代码清单 20-2 解析 JSON 文本以获取 JavaScript 对象 (http://jsbin.com/rexolo/edit?js,console)

```
var xhr = new XMLHttpRequest();
var url = "http://output.jsbin.com/qiwizo.json";
xhr.addEventListener("load", function () {
    var data = JSON.parse(xhr.responseText);//解析 JSON 文本以获取 JavaScript 对象
```

```
        console.log(data);
    });
    xhr.open("GET", url);
    xhr.send();
```

现在，我们脸上的表情应该是一个大大的笑脸。只需几行代码，就可以从世界上任何地方访问数据（托管我们网页的服务器可能存放在任何地方）。一起来拥抱庆祝一下！

在本书中开发的所有应用程序都可以从这个简单的方法中获益：本书所有程序都可以访问 JS Bin 上的数据。但是，程序所生成的 bin 代码（例如 qiwizo）有些难以处理。在下一节，将介绍如何把 XHR 代码打包到 load 函数，以便更加方便地检索数据。

20.1.5　加载 JS Bin 数据——一个好用的函数

本书中的其他几个程序，诸如测验应用程序、My Movie Ratings 页面、健身应用程序的新闻页面和游戏 The Crypt，也都具有在运行时加载数据的能力。为了简化从 JS Bin 加载数据的过程，需要创建一个模块：Bin Data。

首先，将 load 函数添加到 gpwj 命名空间（参见第 19 章）。要在 JS Bin 文件中加载数据，请使用以下格式的代码：

```
function doSomethingWithData (data) {
    //使用数据做些令人惊喜的事
}
gpwj.data.load("qiwizo", doSomethingWithData);
```

以下代码清单显示了程序运行中的 load 函数，输出如图 20-4 所示。



代码清单 20-3　一个用于加载 JS Bin 数据的函数 (http://jsbin.com/suxoge/edit?js,console)

```
(function () {
    "use strict";
    function loadData (bin, callback) {
        var xhr = new XMLHttpRequest();
        var url = "http://output.jsbin.com/" + bin + ".json";  //使用bin代码构
                                                                //建URL
        xhr.addEventListener("load", function () {
            var data = JSON.parse(xhr.responseText);
                callback(data);      //将加载的数据传递给回调函数
        });
        xhr.open("GET", url);
        xhr.send();
    }
    if (window.gpwj === undefined) {
        window.gpwj = {};
    }
    gpwj.data = {
        load: loadData
```

373

```
    };
})();
gpwj.data.load("qiwizo",console.log);    //测试该函数，命令它用加载的数据调用
                                         //console.log
```

在 20.3 节，我们将使用 load 函数在游戏 The Crypt 中获取位置数据。其他几个程序的数据加载版可以在本书网站上获得，网址为 www.room51.co.uk/books/getprogramming/projects/。下面就来对健身应用程序进行升级！

20.1.6　升级健身应用程序

健身应用开发团队看到我们已经完成的基于控制台的工作原型，非常满意。现在，有以下三个更新任务：

（1）更新控制器模块，对用户数据使用新的 load 函数。

（2）使用 script 元素来加载应用程序使用的模块。

（3）添加一行 JavaScript 代码，对应用程序进行初始化。

以下代码清单 20-4 是新的控制器模块代码。程序中使用 loadUser 函数来调用 gpwj.data.load 方法。loadUser 和 log 都是从 init 方法返回的接口。



代码清单 20-4　健身应用控制器 (http://jsbin.com/nudezo/edit?js,console)

```
(function () {
    "use strict";
    var user = null;
    function buildUser (userData) { /*代码清单 14-3 */ }
    function updateUserFromData(userData) {    //从加载的数据中创建一个用户
                                               //并更新显示
        user = buildUser(userData);
        fitnessApp.userView.render(user);
    }
    function loadUser (id) {
        gpwj.data.load(id, updateUserFromData);//加载用户数据并调用回调函数
        return "Loading user details...";
374 }
    function log (sessionDate, duration){//定义一个函数，让用户记录其锻炼情况
            if (user !== null) {
            user.addSession(sessionDate, duration);
            fitnessApp.userView.render(user);
            return "Thanks for logging your session.";
        } else {
            return "Please wait for user details to load.";
        }
    }
    function init (id) {
            loadUser(id);     //为初始用户加载数据
```

```
        return {    //返回一个接口对象
            log: log,
            loadUser: loadUser
        };
    }
    if (window.fitnessApp === undefined) {
        window.fitnessApp = {};
    }
    fitnessApp.init = init;
})();
```

在以上代码中，由于应用程序在运行中加载用户数据，有可能会出现用户在加载数据之前就已经记录了一次锻炼的情况（健身应用程序的用户都超级热情）。为了避免出现上述情况，log 方法要确保先设置用户，user !== null，然后才允许记录。别忘记，null 是一个特殊的 JavaScript 值，通常用于表示该对象是预期的，但是目前找不到或目前尚未赋值。

既然所有部件都准备到位，现在我们就来把它们组装到一起，然后就可以进入实弹射击阶段了。健身应用程序的 4 个模块如图 20-5 所示，可以看到，我们用 Bin Data 模块替换掉了原先提供静态数据的模块。



图 20-5　健身应用程序的各个组成模块

我们不必更改 User 构造函数和视图。下一个代码清单显示用于加载 4 个模块的 script 元素。



代码清单 20-5　健身应用程序（HTML）
(http://jsbin.com/mikigo/edit?html,console)

```
<!--fitnessApp.User -->
<script src="http://output.jsbin.com/fasebo.js"></script>
<!--fitnessApp.userView -->
<script src="http://output.jsbin.com/yapahe.js"></script>    375
<!--fitnessApp.controller -->
<script src="http://output.jsbin.com/nudezo.js"></script>
<!--gpwj.data -->
<script src="http://output.jsbin.com/guzula.json"></script>
```

我们已经加载好 4 个所需模块，下一步就要扣动扳机了。下面，我们来执行更新任务中的第（3）项：添加一行 JavaScript 代码，调用 fitnessApp.init，对应用程序进行初始化。



代码清单 20-6　初始化健身应用程序
(http://jsbin.com/mikigo/edit?js,console)

```
var app = fitnessApp.init("qiwizo");
```

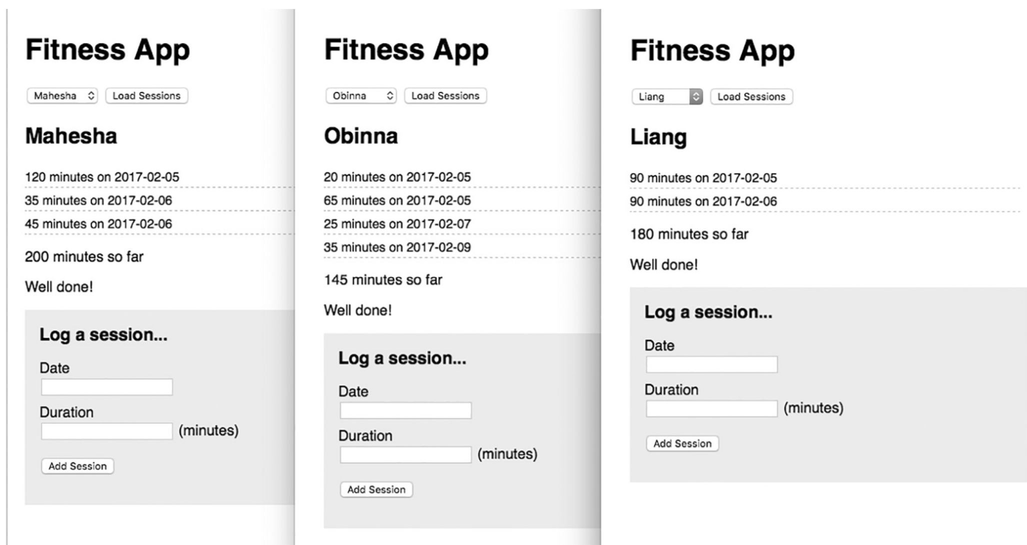
虽然在本章开头的图 20-1 中，JS Bin 被用于显示用户加载和记录运动情况的会话，其

实，JS Bin 是观察应用程序运行情况的最佳场所。

开发团队都对以上应用程序的更新感到很满意，但还是有些遗憾之处。数据到底保存在哪里？本书第三部分一直在讲解如何构造基于 HTML 的应用程序，为什么最后又回到了 JS Bin 控制台？

20.1.7 健身应用程序——下一个目标

图 20-6 显示了这款健身应用程序的预期成果：一个基于 HTML 的健身应用程序，包含下拉列表、文本框和按钮。我们可以从列表中选择某个用户，加载其详细信息并记录新的会话。读者利用在本书中学到的知识，现在能否构建这个应用程序？请读者不必瞻前顾后，勇敢地行动起来！只有亲自动手并努力构建，才能帮助你掌握这些技能。抛开所有压力和顾虑，只需携带勇气和好奇，大胆犯错，勇于求助，即使你没有到达终点线，也会一路学到很多，得到一次很好的 JavaScript 锻炼机会，并练就了一身强壮的编程肌肉。当你亲自尝试之后，就可以偷偷访问一下 http://jsbin.com/vayogu/edit?output，看看各部分组装在一起之后的模样（在 JS Bin 上，别忘记单击 Run 按钮，运行该程序）。



376 图 20-6 使用基于 Web 视图的健身应用程序

如果想要开发一个基于 HTML 数据驱动的应用程序，游戏 The Crypt 是不二之选。

在下一节，先对 JSON 数据格式做一个简要介绍。

20.2 JSON——一种简单的数据格式

JSON 是一种数据格式，不仅易于人类阅读和写入，而且易于计算机解析和生成，是目前在网络上进行数据交换的流行格式。以下是一个用 JSON 编写的日历活动：

```
{
    "title" : "Cooking with Cheese",
```

```
    "date" : "Wed 20 June",
    "location" : "The Kitchen"
}
```

以上格式看似并不陌生，它是基于 JavaScript 的一个子集。属性名称和字符串必须放在双引号里面，属性的值可以是数组或嵌套对象。下面是 6 月份的一些日历数据：

```
{
    "calEvents" : [
        {
            "title" : "Sword Sharpening",
            "date" : "Mon 3 June",
            "location" : "The Crypt"
        },
        {
            "title" : "Team Work Session",
            "date" : "Mon 17 June",
            "location" : "The Crypt"
        },
        {
            "title" : "Cooking with Cheese",
            "date" : "Wed 20 June",
            "location" : "The Kitchen"
        }
    ]
}
```

属性的值还可以是数字、布尔值、null 和 undefined，它们基本可以满足我们所有需要（JSON 实在是太强大了，完整规范请访问 http://json.org。看到该网页所示的 JSON，你可能会惊叹 JSON 是如此简短！——我相信，那些形似火车轨道的图表所产生的高效率一定会让你叹为观止！）。

我们已知，JSON 数据可以作为文本进行传输。下一步，如何将其转换为可以在 JavaScript 程序中使用的对象和数组？ 377

20.2.1　使用 JSON.parse 将 JSON 转换为对象和数组

将 JSON 传递到 JSON.parse 方法，可以把 JSON 文本转换为 JavaScript 对象。然后，就可以使用圆点或方括号运算符访问数据中的属性或元素。如果已经使用 XHR 对象将单个日历事件的请求作为 JSON 发送，则可以将响应转换为如下所示的 JavaScript 对象：

```
var calEvent = JSON.parse(xhr.responseText);
calEvent.title;         // Cooking with Cheese
calEvent.date;          // Wed 20 June
calEvent.location;      // The Kitchen
```

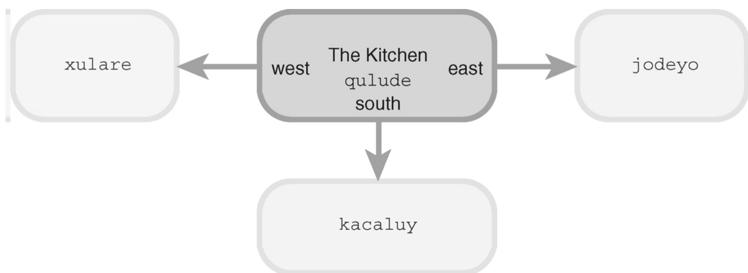
以上是对 JSON 的简短介绍。鉴于整本书是关于 JavaScript 编程，这个简短介绍应该足够了。

下面，我们返回游戏 The Crypt，从一个神秘房间进入另一个神秘房间，按需加载地图数据。

20.3 游戏 The Crypt——根据需要加载地图

游戏 The Crypt 中的探险地点包括古老的墓穴、扁平的飞船和神秘的森林。玩家的每次冒险活动都可能覆盖几十个地点，几乎不可能在一个地方就完成。游戏 The Crypt 并不是一开始就加载整个地图，因为只有当玩家访问某地点时，加载该地点的地图才有意义。游戏可以将每个地点的 JSON 数据存储在 JS Bin 的单独文件中，并在需要时使用 XMLHttpRequest 加载该文件。

图 20-7 只显示了游戏中的第一个位置，在游戏开始时加载。加载的数据应该包括该位置各个出口的 JS Bin 文件代码。例如，The Kitchen 向南出口的数据文件应该包括代码 kacaluy。



378 图 20-7　在游戏开始时，只加载初始位置数据

只有当玩家移动到某地点时，才会加载该地点的数据。图 20-8 显示，在同一幅地图中，当玩家从 The Kitchen 向南移动时，就会加载 The Old Library 的数据。

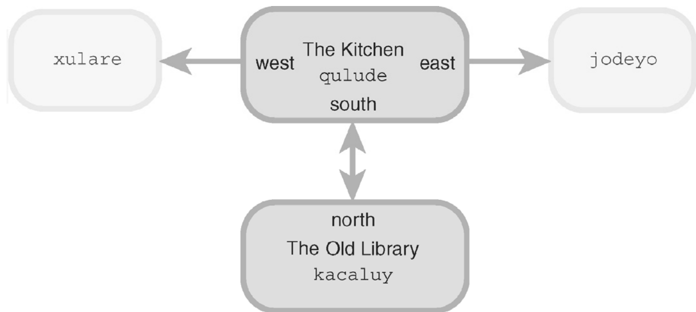


图 20-8　因为玩家移动到 The Old Library，所以该位置的数据已加载

为了实现以上这种逐步加载地图的方式，我们必须确保在地图数据中存有每个位置出口的 JS Bin 文件代码。

20.3.1 使用 JS Bin 文件代码指定出口

假设 The Crypt 游戏从 The Kitchen 开始。首先，游戏只加载 The Kitchen 的数据，而不会加载地图上任何其他位置的数据，这样做使游戏开始时需要加载的数据量大大变少。以下

代码清单 20-7 显示了 The Kitchen 单个位置的数据（为了让读者可以专注于各个 id，省略了挑战部分的代码）。



代码清单 20-7　The Kitchen 的 JSON 数据 (http://output.jsbin.com/qulude.json)

```
{
    "title" : "The Kitchen",
    "id" : "qulude",
    "description" : "You are in a kitchen. There is a disturbing smell.",
    "items" : [ "a piece of cheese" ],
    "exits" : [ //包括每个出口目的地的代码，以便玩家移动到此地时，就加载此地的数据
        {
            "direction" : "south",
            "to" : "The Old Library",
                "id" : "kacaluy"
        },
        {
            "direction" : "west",
            "to" : "The Kitchen Garden",
                "id" : "xulare"
        },
        {
            "direction" : "east",
            "to" : "The Kitchen Cupboard",
                "id" : "jodeyo"
        }
    ]
}
```

379

每个位置都有唯一的 id，与存储该位置数据文件的 JS Bin 代码是一一对应关系。我们可以使用 id 构建一个位置的 URL，如下所示：

```
var url = "http://output.jsbin.com/" + id + ".json";
```

玩家将在地图上来回移动，解决谜题、收集宝藏、击退僵尸和打败豹子。他们有可能多次访问某些地点。下面介绍如何避免程序重复加载相同的数据。

20.3.2　使用缓存——针对每个地点只加载一次

每当使用加载的数据创建一个新地点时，都会使用该地点的 id 作为“键”将其存储在缓存中；再次请求加载该地点时，就可以直接从缓存获取数据。这样可以避免从 JS Bin 中重复加载相同的数据。

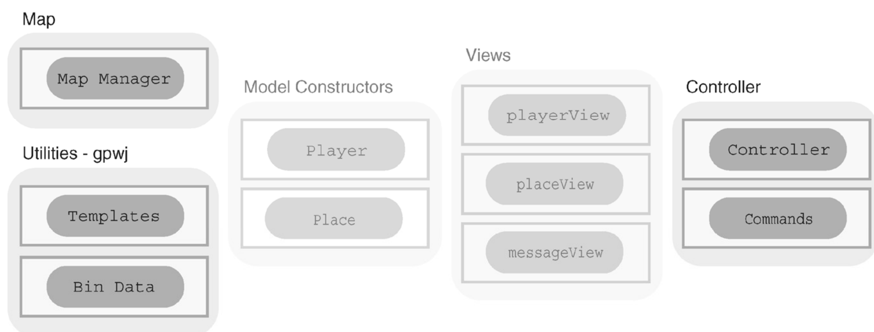
```
var placeStore = {};      //设置一个对象来存储已加载的地点
placesStore[placeData.id] = place;      //使用该地点的 id 作为“键”
```

20.3.3 使用地图管理器替换地图数据模块和地图构建器模块

游戏 The Crypt 的地图构建器用于为整个地图获取数据，创建所有地点模型，然后向地点添加物品、出口和挑战。该地图构建器为整个地图完成了以上工作，就会返回第一个地点，准备开始游戏。现在，我们每次只加载一个地点，就需要一个函数加载单个地点的数据，还需要另一个函数使用已加载的数据构建单个地点模型。

以下代码清单 20-8 是一个新的地图管理器（Map Manager）模块代码，它将替换 The Crypt 游戏早期版本中使用的地图构建器代码（图 20-9）。地图管理器模块在其接口中提供单一方法 loadPlace.Use，可以使用先前定义的回调函数或函数表达式来调用该方法：

```
//使用先前定义的函数作为回调函数
theCrypt.map.loadPlace("qiwizo", doSomethingWithPlace);
// 使用函数表达式作为回调函数
theCrypt.map.loadPlace("qiwizo", function (place) {
    // 使用地点模型
});
```



380 图 20-9 重点显示游戏中的新模块和更新模块



代码清单 20-8 地图管理器 (http://jsbin.com/xesoxu/edit?js)

```
(function () {
    "use strict";
    var placesStore = {};       //使用一个对象作为已加载地点的缓存
    function addPlace (placeData) {
        var place = new theCrypt.Place(   //使用地点数据创建 Place 对象
        placeData.title, placeData.description);
        placesStore[placeData.id] = place; //将新创建的地点缓存在 placesStore
                                           //中，使用其 id 作为“键”
        if (placeData.items !== undefined) {
            placeData.items.forEach(place.addItem);
        }
            if (placeData.exits !== undefined) {
            placeData.exits.forEach(function (exit) {
                place.addExit(exit.direction, exit.id);    //为指定方向的出
                                                           //口添加地点 id
```

```
                place.addChallenge(exit.direction, exit.challenge);
            });
        }
        return place;
    }
    function loadPlace (id, callback) {   //函数包括一个 callback 形参，以便
                                          //使用加载地点来调用该函数
        var place = placesStore[id];      //检查缓存,看是否有所请求的地点
        if (place === undefined) {        //仅加载缓存中没有的地点数据
            gpwj.data.load(id, function (placeData) {     //使用 Bin 数据模
                                                          //块加载数据
                var place = addPlace(placeData);
                callback(place);       //使用从数据中新构建的地点调用回调函数
            });
        } else {
                callback(place);       //如果某地点已经加载过，请使用缓存中的该地
                                       //点调用回调函数
        }
    }
    if (window.theCrypt === undefined) {
        window.theCrypt = {};
    }
    theCrypt.map = {
        loadPlace: loadPlace
    };
})();
```

381

地图管理器由两个函数组成。下面依次探讨这两个函数。

1. 把一个地点添加到存储中

addPlace 函数首先使用 Place 构造函数根据数据创建出一个地点模型，然后为其添加物品、出口和挑战，再将该地点存储在缓存中，最后返回新的地点模型。

```
function addPlace (placeData) {
    // 创建一个地点模型
    // 将该地点存入缓存中
    // 添加物品
    // 添加出口和挑战
    // 返回新的地点模型
}
```

关于这个函数，除了缓存部分，其他内容在前面章节中都已经讲过，不再赘述。

2. 为单个地点加载数据

当调用 loadPlace 函数时，需要为其提供欲加载地点的 id 和一个回调函数。使用该 id 的地点模型作为实参调用该回调函数，如下所示：

```
callback(place);
```

loadPlace 函数从哪个渠道获得地点模型？首先，它尝试从缓存中获取该地点；如果在缓存中找不到，就从 JS Bin 中加载该地点模型。代码如下：

```
var place = placesStore[id]; // 尝试从缓存中获得该地点
if (place === undefined) {
    // 该地点不在缓存中
    // 从 JS Bin 加载该地点数据
    // 将该地点传递给回调函数
    callback(place);
} else {
    //该地点在缓存中
    //将该地点传递给回调函数
    callback(place);
}
```

如果 loadPlace 函数必须加载数据，就会使用 Bin Data 模块的 load 函数从 JS Bin 中获取数据。既然该地点不在缓存中，代码就会将地点数据传递到 addPlace，以便先创建和存储该
382 模型，再将新的地点模型传递到回调函数。

```
gpwj.data.load(id, function (placeData) {
    var place = addPlace(placeData);     //创建和存储一个地点模型
    callback(place);     //将新的地点模型传递到回调函数
});
```

现在，我们已经掌握了数据加载知识，下面将把这些知识应用到游戏控制器中，先加载一个初始地点，再随玩家移动到一个新的地点。

20.3.4 更新游戏控制器

为了使用新的地图管理器，需要对游戏控制器中的两个方法进行更新。以前，我们使用 init 方法调用 buildMap，构建一整张地图。现在 init 方法将调用 loadPlace，仅加载游戏中的第一个地点。因为现在不再预加载地点，因此 go 方法需要调用 loadPlace，从缓存中查找一个地点或者从 JS Bin 加载一个地点。

以下代码清单显示了对游戏控制器中的两个方法进行的更新。



代码清单 20-9 使用地图管理器的游戏控制器 (http://jsbin.com/vezaza/edit?js)

```
(function () {
    "use strict";
    var player;
    var inPlay = false;
    function init (firstPlaceId, playerName) {
        theCrypt.map.loadPlace(      //使用 loadPlace 加载游戏中的第一个地点并
                                     //运行 init 代码
            firstPlaceId, function (firstPlace) {
                player = new theCrypt.Player(playerName, 50);
```

```
                player.addItem("The Sword of Doom");
                player.setPlace(firstPlace);
                inPlay = true;
                render();
            });
        }
        function go (direction) {
            if (inPlay) {
                /* 声明变量 */
                if (destination === undefined) {
                    renderMessage("There is no exit in that direction");
                } else {
                    if ((challenge === undefined) || challenge.complete) {
                        theCrypt.map.loadPlace(    //加载玩家的新地点
                            destination, function (place) {
                            player.setPlace(place); //加载后，将该地点设置为玩家的位
                                                    //置并更新显示
                            render();
                            });
                    } else {
                        /* 应用挑战 */
                    }
                }
            } else {
                renderMessage("The game is over!");
            }
        }
        /* 其他函数 */
        window.game = { … };    //分配公共接口
    })();
```

383

在加载开始位置之前，init 方法几乎无法工作。因此，init 方法的大部分代码是放在回调函数内部传递给 loadPlace。go 方法也是使用 loadPlace 获取位置，然后将其设置为玩家的新位置。

20.3.5　构建游戏页面

万事俱备，现在我们只需导入各个模块，再添加几行 JavaScript，就可以对游戏进行初始化了。

以下代码清单显示了该游戏的完整 HTML。其中消息 div 和命令文本框的位置已经移至地点和玩家部分的上方。



代码清单 20-10　The Crypt (HTML)
(http://jsbin.com/cujibok/edit?html,output)

```
<!DOCTYPE html>
<html>
```

```
<head>
   <meta charset="utf-8">
   <title>The Crypt</title>
</head>
<body>
   <h1>The Crypt</h1>
   <div id="messages" class="hidden"></div>
   <div id="controls">
      <input type="text" id="txtCommand" />
      <input type="button" id="btnCommand" value="Make it so" />
   </div>
   <div id="views">
      <div id="place"></div>
      <div id="player"></div>
   </div>
   <!-- 各个模板-->
   <script type="text/x-template" id="itemTemplate">
      <li>{{item}}</li>
384   </script>
   <script type="text/x-template" id="playerTemplate">
      <h3>{{name}}</h3>
      <p>{{health}}</p>
      <ol id="playerItems"></ol>
   </script>
   <script type="text/x-template" id="placeTemplate">
      <h3>{{title}}</h3>
      <p>{{description}}</p>
      <div class="placePanel">
         <h4>Items</h4>
         <ol id="placeItems"></ol>
      </div>
      <div class="placePanel">
         <h4>Exits</h4>
         <ol id="placeExits"></ol>
      </div>
   </script>
   <script type="text/x-template" id="messageTemplate">
      <p>*** {{message}} ***</p>
   </script>
   <!-- 各个模块 -->
   <!-- gpwj.templates -->
   <script src="http://output.jsbin.com/pugase.js"></script>
   <!-- gpwj.data -->
   <script src="http://output.jsbin.com/guzula.js"></script>
   <!-- 地点构造器 -->
   <script src="http://output.jsbin.com/vuwave.js"></script>
```

```
    <!-- 玩家构造器 -->
    <script src="http://output.jsbin.com/nonari.js"></script>
    <!-- 玩家视图 -->
    <script src="http://output.jsbin.com/suyona.js"></script>
    <!-- 地点视图 w -->
    <script src="http://output.jsbin.com/yoquna.js"></script>
    <!-- 消息视图-->
    <script src="http://output.jsbin.com/jojeyo.js"></script>
    <!-- 地图管理器 -->
    <script src="http://output.jsbin.com/xesoxu.js"></script>
    <!-- 游戏控制器 -->
    <script src="http://output.jsbin.com/vezaza.js"></script>
    <!-- 网页控件 -->
    <script src="http://output.jsbin.com/xoyasi.js"></script>
</body>
</html>
```

以下代码清单显示了 JavaScript 初始化代码，为游戏始发地的数据文件指定了 JS Bin 代码。 385



代码清单 20-11　The Crypt (http://jsbin.com/cujibok/edit?js,output)

```
var playerName = "Kandra";
var firstPlaceId = "vitewib";        //为地图中第一个地点指定 JS Bin 文件的 id
game.init(firstPlaceId, playerName);
```

20.3.6　进入游戏 The Crypt

太棒了！游戏的各个模板与其他部分的 HTML 完美结合。以后，如果需要改变设计，就不必面对 JavaScript、数据和 HTML 三者交织在一起的混乱局面，而是可以很容易地找到需要修改的模板。对开发者而言，这样的程序更易于维护，更不易出错，也意味着给我们带来更大的快乐。胜利，胜利，胜利！

现在就可以开始玩游戏了！你唯一的目标就是在游戏中求生存，在 JS Bin 上考验你的力量、技能和耐力。祝你好运！游戏网址是 http://output.jsbin.com/cujibok。

20.4　本章小结

- 使用 XMLHttpRequest 对象为网页加载资源，而不需重新加载整个页面。
- 使用 JSON（一种简洁易懂的 JavaScript 数据交换格式）传输应用程序的数据。
- 为了加载数据，首先创建 XMLHttpRequest 对象，然后为 load 事件设置事件监听器，再使用资源的 URL 调用 open 方法，最后调用 send 方法：

```
var xhr = new XMLHttpRequest();
xhr.addEventListener("load", function () {
```

```
    // 使用 xhr.responseText
});
xhr.open("GET", url);
xhr.send();
```

- 当 load 事件被触发时，通过 XHR 对象的 responseText 属性访问 JSON 数据。
- 将 JSON 数据字符串传递到 JSON.parse 方法，转换为 JavaScript 对象：

```
var data = JSON.parse(xhr.responseText);
```

- 使用 JavaScript 对象作为缓存。首先将加载过的数据或模型放入缓存，然后在使用
386 XHR 之前检查缓存。

第 21 章　结语：使用 JavaScript 编程

本章内容包括：

- 深化成果
- 在本地处理文件
- 图书和资源

如果读者提问，如何使用 JavaScript 编程？

我的回答是，你们正在使用 JavaScript 编程。

如果你真的想要理解编程，就必须行动起来，动手编写程序。通过阅读书籍和思索代码来学习编程也许轻松有趣，也能够让你增长知识，但是只有亲自解决问题和编写代码才能真正提高编程技能、积累经验并增强抗挫能力。请读者放心，你并不是孤军奋战。如果你暂时困在一个项目上不得进展，或者是对一些代码百思不得其解，在本章中，你将看到求助的平台。下面，我们首先介绍如何在自己本地计算机上保存的网页中使用 JavaScript，而不是在 JS Bin 上使用 JavaScript。

21.1　在本地处理文件

JS Bin 和其他在线编码网站是很好的实验场所，可供我们实验新的想法，还可以检验代码的运行情况。但是，现在我们想摆脱沙盒环境，创建自己的网站。本节将介绍如何在本地（即自己的计算机上）编写和保存文件，并在浏览器中打开文件。 387

21.1.1　编写代码

JavaScript 和 HTML 文件都是文本文件，可以在任何文本编辑器中对其进行编写。Windows 上的 Notepad（记事本）和 OS X 上的 TextEdit 尽管都非常简单，却都可以胜任此项工作。更高级的文本编辑器支持语法高亮，可以使用不同颜色将关键字从变量、参数、字符串等其他内容中区分出来，可以对输入内容提供建议，还可以为快速插入提供建议。目前比较流行的编辑器有 Sublime Text、BBEdit、Notepad ++、Atom 和 Emacs。

另外，还有集成开发环境（IDE），可以提供更多的项目管理工具，例如相互协作；跟踪各版本；合并、缩小和压缩文件等。目前比较流行 Visual Studio、Dreamweaver、Eclipse 和 WebStorm。 图 21-1 显示了三个编辑器：Notepad、Notepad++和 WebStorm，它们对项目的支撑水准依次提高，也就是说，这三者当中，Notepad 对项目的支撑水准最低，WebStorm 最高。

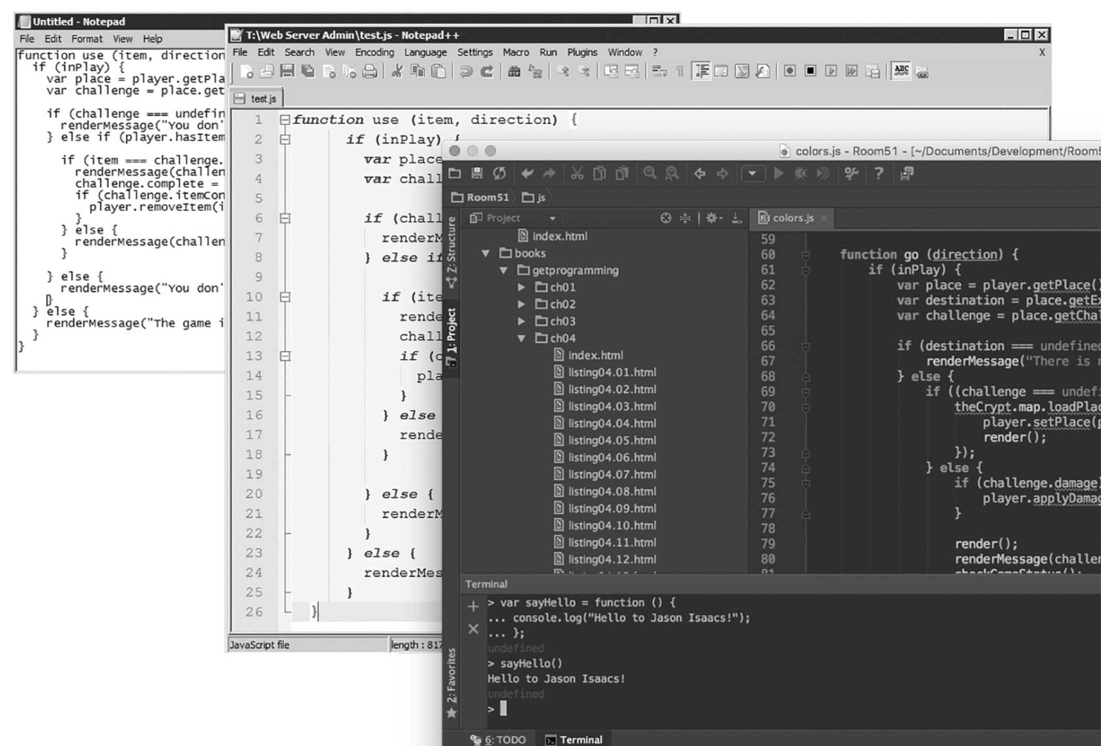


图 21-1　Notepad、Notepad ++和 WebStorm

21.1.2　保存文件

本节讲解如何有条理地保存文件，即如何使用合理的文件夹将文件分类或按照不同的子项目进行保存。图 21-2 显示了 theCrypt 文件夹内的文件存放结构。在该文件夹内，分别使不同的文件夹存放各类文件：样式表存于 css，JavaScript 文件存于 js，游戏地图存于 maps，
388 主要的 HTML 文件 theCrypt.html 位于该项目的根目录中。在 GitHub 代码库中可以找到本书所有应用程序的文件夹，网址是 https://github.com/jrlarsen/GetProgramming。

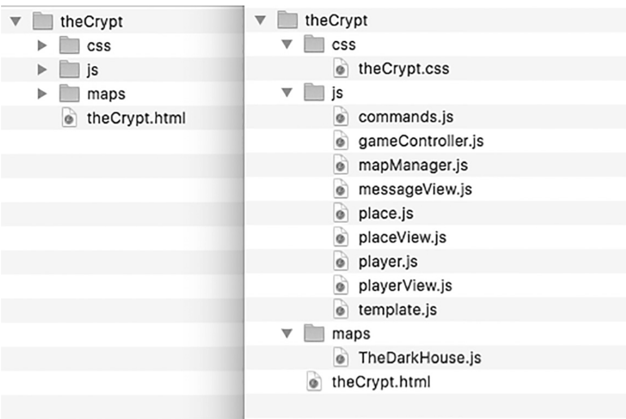


图 21-2　theCrypt 文件夹内的文件存放结构

文件名称应该表明该文件的目的。虽然 JS Bin 会为文件分配随机名称，但是这并不意味着我们应该这样做。以下代码清单显示了游戏的 HTML 文件，其中包含用于加载 JavaScript 的 script 元素，在 head 部分中有一个 link 元素，用于加载为页面设置样式的 CSS 文件，该代码清单中省略了那些无任何更改的模板。



代码清单 21-1　theCrypt.html 中的 script 元素

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <link rel="stylesheet" href="css/theCrypt.css" /><!-- 加载 CSS 文件，
                                                        设置页面的样式 -->
<title>The Crypt</title>
</head>
<body>
<h1>The Crypt</h1>
<div id="player"></div>
<div id="place"></div>
<div id="messages"></div>
<p id="controls">
   <input type="text" id="txtCommand">
   <button id="btnCommand">Make it so</button>
</p>
```

389

```
   <!-- 模板 -->
   <!-- 无改变部分请参阅前面章节 -->
   <!-- 模块 -->
   <script src="maps/TheDarkHouse.js"></script>   <!-- 可以使用相对路径
                                                   加载 script 文件 -->
   <script src="js/mapManager.js"></script>
   <script src="js/template.js"></script>
   <script src="js/player.js"></script>
   <script src="js/place.js"></script>
   <script src="js/playerView.js"></script>
   <script src="js/placeView.js"></script>
   <script src="js/messageView.js"></script>
   <script src="js/gameController.js"></script>
   <script src="js/commands.js"></script>
</body>
</html>
```

在 src 属性中使用的路径告诉浏览器在哪里能够找到与 theCrypt.html 相关的加载文件。例如，maps/TheDarkHouse.js 表示从 Crypt.html 所在的文件夹开始，先找到 maps 文件夹，然后在该文件夹中找到名为 TheDarkHouse.js 的文件。

21.1.3　在浏览器中打开页面

在浏览器菜单中，选择 File（文件），再选择 Open（打开），然后浏览本地计算机的文

件系统，找到并单击 theCrypt.html，就会在浏览器中打开该页面。

21.1.4 合并和压缩文件

如果在 Web 服务器上托管某个项目以供用户通过互联网访问，那么最好将所有的 JavaScript 模块都放在一个文件中。原因在于，针对每个 script 元素，浏览器都要向服务器请求文件，该文件由 src 属性指定，如果需要加载所有文件才能使该应用程序运行，那么加载一个大文件比加载很多小文件耗时更少。

我们可以在 JS Bin 上看到，The Crypt 的所有 JavaScript 包含在一个 bin 中：http://jsbin.com/xewozi/edit?js,output。

在开发应用程序时，使用各个单独的文件有益于明确、重用和灵活性。但是，在发布应用程序时，合并文件会有助于更快地加载代码。开发者可以手动截取代码并将其复制到单个文件，也可以使用工具来帮忙。以下工具可能对初学者来说有些过于先进，如果读者有兴趣，可以关注 Grunt、Gulp、Browserify 和 CodeKit。

这些工具还可以将代码压缩成一个较小的文件。当浏览器运行 JavaScript 时，并不需要空格和换行符，它们只是用于帮助人类阅读和理解代码；也不需要合理命名的变量，例如变量 a、b、c 与变量 loadData、checkGameStatus、renderMessage 完全等效。因此，这些工具会剥离空格并重新命名变量以创建一个更小的文件，该文件的执行效果并无任何改变。下面两
390 个代码清单显示了相同的代码，取自 The Crypt 控制器模块的 use 函数。



代码清单 21-2　use 函数（缩小版）

```
function r(e,a){if(s){var o=l.getPlace(),i=o.getChallenge(a);void
0===i||i.complete===!0?t("You don't need to use that
there"):l.hasItem(e)?e===i.requires?(t(i.success),i.complete=!0,i.itemConsume
d&&l.removeItem(e)):t(i.failure):t("You don't have that item")}else t("The
game is over!")}
```



代码清单 21-3　use 函数

```
function use (item, direction) {
   if (inPlay) {
      var place = player.getPlace();
      var challenge = place.getChallenge(direction);
      if (challenge === undefined || challenge.complete === true) {
         renderMessage("You don't need to use that there");
      } else if (player.hasItem(item)) {
         if (item === challenge.requires) {
            renderMessage(challenge.success);
            challenge.complete = true;
            if (challenge.itemConsumed) {
               player.removeItem(item);
            }
         } else {
            renderMessage(challenge.failure);
```

```
                }
            } else {
                renderMessage("You don't have that item");
            }
        } else {
            renderMessage("The game is over!");
        }
    }
```

通过对比以上两个代码清单，读者应该可以理解压缩文件的意义所在。

21.2　获取帮助

现在，你基本上已经通读完本书，可以看出你很重视有指导、有组织的学习。但是，这并不意味着你已经真正上路并走上正轨。我建议读者能够深入地挖掘资源，寻找问题，并就具体问题寻求帮助。本书推荐一个很好的求助网站，供你参考：Mozilla Developer Network（https://developer.mozilla.org）。该网站上有许多 HTML、CSS 和 JavaScript 示例。如果你在寻求具体问题的解决方法，请加入 stackoverflow 论坛（http://stackoverflow.com），该论坛的成员通常会快速解答问题，关于 JavaScript 的问题已经突破一百万个，在众多编程语言中，JavaScript 率先达到一百万！ 391

21.3　后续推荐

本书是一本编程入门书。第一部分介绍了 JavaScript 语言的基础知识，第二和第三部分组织构建了一些稍大的项目，为将来做更大的项目做好准备。撰写本书的目的并非是为读者提供一本完整的编程参考书，而是为读者提供一次编程的实践体验。建议读者在阅读本书之后，继续深入学习……

21.3.1　本书网站

在本书网站上可以看到持续更新的文章、教程、视频、资源、示例以及其他参考网站的链接。网址如下：www.room51.co.uk/books/getprogramming/index.html。

该网站内容还包括在本书没有介绍的一些主题，例如原型、继承、使用 this、Node.js 和 JavaScript 的最新增补内容，另外还有对本书中应用程序的改进指南，例如测验应用程序、健身应用程序、健身应用程序新闻以及电影评分应用程序等。

21.3.2　图书

市场上有很多关于 JavaScript 编程的书籍，下面推荐我阅读过并非常喜欢的两本书。第一本可以作为本书的后续书籍，第二本是对 JavaScript 语言更加深入的探索：

- Eloquent JavaScript: A Modern Introduction to Programming (No Starch Press; 2nd ed., 2014) by Marijn Haverbeke
- Professional JavaScript for Web Developers (Wrox; 3rd ed., 2012) by Nicholas C. Zakas

21.3.3 网站

许多网站将视频教程与互动练习相结合。我本人特别喜欢 Code School，http://codeschool.com。该网站人员还建立了一个不错的 JavaScript 介绍性网站，http://javascript.com。

21.3.4 实践——熟能生巧

请读者继续实践，到 JS Bin 去调试程序。本书的网站上有一些对编程项目的建议，如果有不理解之处，请随时在 Manning.com 的书籍论坛上发布问题（https://forums.manning.com/forums/get-programming-with-javascript），或者在 stackoverflow.com 加入社区论坛。

希望读者能够边读书边实践，在 JS Bin 上实验代码，大胆猜测并验证你的各种想法。让
392 我们保持好奇心，勇敢地进行 JavaScript 开发实战吧！

21.4 本章小结

- 使用文本编辑器编写并保存 JavaScript 和 HTML 文件。
- 使用文件夹和适当的文件名来整理一个项目的文件。
- 在浏览器中打开 HTML 文件。
- 在 Mozilla Developer Network 和 stackoverflow.com 了解更多信息，并获取帮助。
- 充分利用本书网站上的资源。
- 实践，实践，再实践。
- 393 请读者在探索中保持好奇心、冒险劲、适应力和忍耐力。祝各位探险顺利！

索　　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

■ 符号

!== operator（!==运算符）, 209
* operator（*运算符）, 21, 42
+ symbol（+符号）, 28
+= operator（+=运算符）, 98, 110, 126, 154
<= operator（<=运算符）, 209, 289
= operator（=运算符）, 74
<= symbol（<=符号）, 204
= symbol（=符号）, 18
=== operator（===运算符）, 199–200, 209
/ operator（/运算符）, 42
< operator（<运算符）, 209
> operator（<运算符）, 209
>= operator（>=运算符）, 209
|| symbol（||符号）, 293

■ A

abstraction（抽象）, 25
add function（add函数）, 70–71, 73–74
addAge function（addAge函数）, 151
addChallenge method（addChallenge方法）, 258
addEventListener method（addEventListener方法）, 326, 330, 372
addExit method（addExit方法）, 158, 175–176, 254, 261
addExits method（addExits方法）, 139
addItem method（addItem方法）, 137, 254, 272
addItems method（addItems方法）, 139
addMoon method（addMoon方法）, 129
addPlace method（addPlace方法）, 382
addSession method（addSession方法）, 250–251, 265
alert function（alert方法）, 330
answer variable（answer变量）, 174
app.loadUser method（app.loadUser方法）, 368
app.log method（app.log方法）, 281, 284
applyDamage method（applyDamage 方法）, 270
area property（area 属性）, 85
arguments（参数/实参）
  determining return value using（使用参数来确定返回值）, 72–75
  passing multiple arguments to function（将多个实参传递给一个函数）, 63–64
  passing one argument to function（将一个实参传递给一个函数）, 59–62
  using objects as（使用对象作为实参）, 84–87
    accessing properties of object argument（访问对象参数的属性）, 84–85
    adding properties to object argument（增加对象参数的属性）, 85–87
  vs. parameters（形参）, 62
arrays（数组）, 104–121
  array methods（数组方法）, 110–117
    adding and removing elements（添加和删除元素）, 111
    forEach method（forEach 方法）, 113–117
    slice method（slice 方法）, 112
    splice method（splice 方法）, 112–113
  converting JSON data into objects and arrays with（将 JSON 转换为对象和数组）, 378
  creating（创建）, 105–106
  displaying data from（显示数据）, 311–314
  elements of, accessing（元素，访问）, 106–110
  player items array（玩家的物品数组）, 118–121
assignment operator（赋值运算符）, 18, 74
attempted property（attempted 属性）, 34
{{author}} placeholder（{{author}}占位符）, 348, 350
author property（author 属性）, 349

■ B

backslash character（反斜杠符）, 36
between function, using in guessing game（between 函数，用于猜数字游戏）, 231–232
bins, creating（bins，创建）, 225, 227
blank function（blank 函数）, 91–92
blogs, writing（博客）, 35
body element（body 元素）, 333
body key（body 键）, 354
body property（body 属性）, 349
body tag（body 标签）, 308, 319
book variable（book 变量）, 30
bookAuthor variable（bookAuthor 变量）, 28
bookTitle variable（bookTitle 变量）, 28
boolean values（布尔值）, 200
box method（box 方法）, 100
bracket notation（括号运算符）, 147–168
  using square brackets instead of dots（使用方括号代替圆点）, 148–155
    people's names as keys（人们的名字作为键）, 150–152
    word counts（单词统计）, 152–155
  using with Crypt（在 The Crypt 中使用）
    game controls（游戏控制）, 164–167
    linking places via exits（通过出口连接各个场所）, 155–163
break statement（break 语句）, 338
browser support（浏览器的兼容性）, 15
browser, opening local files in（浏览器，打开本地文件）, 390
buildExits function（buildExits 函数）, 259–261
buildMap function（buildMap 函数）, 258–259, 261–262, 288, 383
buildPlace function（buildPlace 函数）, 259
buildPlaces function（buildPlaces 函数）, 261
buildPlanet function（buildPlanet 函数）, 87, 89, 123–127, 129
buildUser function（buildUser 函数）, 252–253
buttons（按钮）, 324–327
  adding to page（添加到页面）, 324–325
  listening for clicks（监听按钮点击）, 325–327, 339–340
  writing functions to update greeting（使用函数更新问候语）, 325

■ C

cache, Crypt game（缓存，The Crypt 游戏）, 380
calculateSizes function（calculateSizes 函数）, 85–87
calculateTax function（calculateTax 函数）, 50
calendars, creating（创建日历）, 35–36
callback function（回调函数）, 370
calling functions（调用函数）, 46–48
callOutCharge variable（callOutCharge 变量）, 21
camel casing（骆驼命名法）, 23
Cascading Style Sheets. See CSS（级联样式表。见 CSS）
chaining calls, replacing one string with another（连续调用，用一个字符串替换另一个）, 348–349
challenges object（challenges 对象）, 258
challenges, Crypt game（挑战，在 The Crypt 游戏中）, 291–292
character argument（character 参数）, 99
checkAnswer function（checkAnswer 函数）, 214
checkGameStatus（checkGameStatus 函数）, 289, 390
clear method（clear 方法）, 165, 321, 340
climbCount（climbCount 变量）, 182
closing tags（结束标签）, 304
CMS (content management system) （CMS（内容管理系统））, 344–345, 358
code on demand（按需代码）, 55
coercion（强制转换）, 209
collisions（冲突）, 234–238
  using namespaces to minimize（使用命名空间来最小化）, 237–238
  variable collisions（变量冲突）, 236–237
Commands module（命令模块）, 335, 340, 342
commands. See player commands（命令，见玩家命令）
commandWords array（commandWords 数组）, 337
comparison operators, else if statement（比较运算符，else if 语句）, 208–209

complete property（complete 属性）, 291
concatenating local files（合并本地文件）, 390–391
conditions（条件）, 198–220
checking answers in quiz app（检查测验应用程序中的答案）, 210–214
checking player's answer（检查玩家的答案）, 213
displaying question（显示问题）, 212–213
handling player's answer（处理玩家答案）, 214
moving to next question（显示下一个问题）, 213
multiple declarations with single var keyword（使用单个 var 关键字声明多个变量。）, 211–212
returning interface object（返回接口对象）, 214
conditional execution of code（条件执行代码）, 199–204
else clause（else 子句）, 200–202
hiding secret number inside functions（在函数中隐藏密码数字）, 202–204
if statement（if 语句）, 200
strict equality operator（严格相等运算符）, 199–200
Crypt-checking user input（在 The Crypt 游戏中检查用户输入）, 214–220
error messages（错误信息）, 216–217
go method（go 方法）, 215–216
using if statement to avoid problems（使用 if 语句来避免问题）, 217–220
else if statement（else if 语句）, 206–209
generating random numbers withMath.random() method（使用 Math.random()方法生成随机数）, 204–206
Console panel（Console 控制面板）, 7, 224, 228–229
console prompt（控制台提示符）, 75–77
calling functions at（调用函数在）, 75–76
declaring new variables at（声明新变量在）, 76–77
console.error method（console.error 方法）, 278
console.log function（console.log 函数）, 8, 19, 41, 75, 106, 109, 152,172, 316
constructors（构造函数）, 122–146
creating multiple independent counters with（创建多个独立计数器）, 182–183
examples of（举例）, 132–134
using to build objects（构建对象）, 127–132
overview（概述）, 127–130
telling objects apart with instanceof operator（使用 instanceof 运算符来区分对象）, 131–132
world building-making use of Planet constructor（使用 Planet 构造函数创建一个新世界）, 130–131
using with Crypt（使用 Crypt）
creating Place objects（创建 Place 对象）, 134–140
streamlining player creation（简化玩家创建代码）, 140–146
content management system. See CMS（内容管理系统。见 CMS）
Controller module（Controller 控制器模块）, 284
controllers（控制器）, 280–298
Crypt game（The Crypt 游戏）, 284–286
approaching controller code（走进控制器代码）, 286
controller code structure（控制器代码结构）, 287
get function（get 函数）, 290–291
go function（go 函数）, 292–293
initializing game（初始化游戏）, 285, 288
listing properties of challenge（列表说明挑战的属性）, 291–292
monitoring player health（监视玩家健康）, 288–289
overview（概述）, 285–286
running game（运行游戏）, 296–297
updating display-functions that use view modules（更新显示——使用视图模块的函数）, 289
use function（use 函数）, 294–296
user input（用户输入）, 286
fitness app（健身应用程序）, 281–284
controls（控件）, 323–342
buttons（按钮）, 324–327
adding to page（添加到页面）, 324–325
listening for clicks（监听按钮点击）, 325–327
writing functions to update greeting（使用函数更新问候语）, 325
Crypt game（The Crypt 游戏）, 334–342
select element（选择元素）, 327–330
adding to page（添加到页面）, 328–329
rateMovie function（rateMovie 函数）, 329–330
text boxes（文本框）, 330–334
adding to page（添加到页面）, 331
adding unordered list to display comments（添加无序列表以显示注释）, 332

CSS 334（级联样式表）
getting references to new elements（获取对新元素的引用）, 332
rateMovie function（rateMovie 函数）, 332–334
correct property（correct 属性）, 34
costPerHour variable（costPerHour 变量）, 21
counter variable（counter 变量）, 179–181
countUpBy1 function（countUpBy1 函数）, 180
Crypt game（The Crypt 游戏）, 11–14, 268–277, 314–322, 378–386
building（构建）, 12–14
building game page（构建游戏页面）, 384–385
building player information strings（构建玩家信息字符串）, 77–82
for player's name, health, and location（玩家的名字、健康和位置）, 78–79
function for player information（用于显示玩家信息的函数）, 79–82
cache（缓存）, 380
checking user input（检查用户输入）, 214–220
error messages（错误信息）, 216–217
go method（go 方法）, 215–216
using if statement to avoid problems（使用 if 语句以避免问题）, 217–220
controllers（控制器）, 284–286
approaching controller code（走进控制器代码）, 286
controller code structure（控制器代码结构）, 287
get function（get 函数）, 290–291
go function（go 函数）, 292–293
initializing game（初始化游戏）, 285, 288
listing properties of challenge（列表说明挑战的属性）, 291–292
monitoring player health（监视玩家健康）, 288–289
overview（概述）, 285–286
running game（运行游戏）, 296–297
updating display-functions that use view modules（更新显示——使用视图模块的函数）, 289
use function（use 函数）, 294–296
user input（用户输入）, 286
CSS（级联样式表）, 319–320
displaying player information（显示玩家信息）, 54–56, 64–69
health（健康）, 66–67
location（位置）s, 67–68
names（名字）, 65–66
hiding place info（隐藏地点信息）, 191–193
hiding player info（隐藏玩家信息）, 187–191
JavaScript's strict mode（JavaScript 严格模式）, 318
map data（地图数据）, 253–263
adding challenges to（向 TheCrypt 地图数据添加挑战）, 256–257
overview（概述）, 255–256
running game（运行游戏）, 262–263
updating Place constructor（更新 Place 构造函数）, 258
using to build game map（用于构建游戏地图）, 258–262
Map Manager module（地图管理器模块）, 380–384
adding place to store（添加地点到存储中）, 382
loading data for single place（为单个地点加载数据）, 382–383
updating game controller to use（更新游戏控制器）, 383–384
message view（消息视图）, 320
modules（模块）
listings（代码清单）, 317–318
loading（加载）, 318–319
render method（render 方法）, 315–316
organizing code into modules（将代码分解成模块）, 242–247
place views（place 视图）, 274–277
model（模型）, 274–276
view（视图）, 276–277
placeholders, adding（占位符，加载）, 318–319
player commands via text box（玩家通过文本框发号命令）, 334–342
adding controls to page（向页面添加控件）, 335–336
Commands module（命令模块）, 340–342
join command（join 命令）, 336–338
listening for button clicks（监听按钮的点击）, 339–340
mapping text box entries to game commands（将文本框内容映射到游戏命令）, 336
pop command（pop 命令）, 336–338
shift command（shift 命令）, 336–338

split command（split 命令）, 336–338
switch block（switch 结构）, 338–339
player items array（玩家的物品数组）, 118–121
player object in（玩家对象）, 37–39
player objects as arguments（玩家对象作为参数）, 101–103
player variables（玩家变量）, 24–26
player views（玩家视图）, 269–274
model（模型）, 270–272
view（视图）, 272–274
playing 11（运行）, 320
specifying exits with JS Bin file codes（用 JS Bin 文件代码指定出口）, 379–380
user interaction（用户交互）, 193–197
using bracket notation with（使用括号运算符）, 155–163
using constructors with creating Place objects（使用创建 Place 对象的构造函数）, 134–140
streamlining player creation（简化玩家创建代码）, 140–146
Crypt game (continued)（The Crypt 游戏）
views, 360–366（视图）
creating HTML templates for（创建 HTML 模板）, 361–362
updating to use new templates（针对新模板更新视图）, 362–365
CSS (Cascading Style Sheets)（级联样式表）
Crypt game（The Crypt 游戏）, 319–320
overview（概述）, 302
text boxes（文本框）, 334
CSS panel, JS Bin（CSS 面板）, 7
curly braces（花括号）, 44, 351

■ D

damage property（damage 属性）, 257, 291
dasher function（dasher 函数）, 234–236, 238
data loading（数据加载）, 367–386
Crypt game（The Crypt 游戏）, 378–386
building game page（构建游戏页面）, 384–385
cache（缓存）, 380
Map Manager module（地图管理器模块）, 380–384
specifying exits with JS Bin file codes（用 JS Bin 文件代码指定出）, 379–380
fitness app（健身应用程序）, 368–377
building（构建）, 374–376
JS Bin data（JS Bin 数据）, 373–374
locating user data（加载用户数据）, 368–370
overview（概述）, 370
parsing XHR response with JSON.parse（使用 JSON.parse 解析 XHR 响应）, 372–373
XMLHttpRequest constructor（XMLHttpRequest 构造函数）, 370–372
JSON data（JSON 数据）, 377–378
data object, fitness app（data 对象，健身应用程序）, 251–252
data. See displaying data（data，见显示数据）
dataArray function（dataArray 函数）, 356
dataObject.name（dataObject.name）, 251
dateString function（dateString 函数）, 134
dayInWeek variable（dayInWeek 变量）, 108
defining functions（定义函数）
new functions（新函数）, 44–45
using function expressions and function declarations（使用函数表达式和函数声明）, 45–46
dependencies（依赖）, 227
description property（description 属性）, 135
destination argument（destination 参数）, 190
destinations variable（destinations 变量）, 175, 216
direction parameter（direction 形参）, 215
direction string（direction 字符串）, 158
displaying data（显示数据）, 264–279
fitness app（健身应用程序）, 265–268
creating views（创建视图）, 266–267
using modules to switch views（使用模块切换视图）, 267–268
from array（数组之中）, 311–314
message view（message 视图）, 278
place views（place 视图）, 274–277
player views（player 视图）, 269–274
displayMenu function（displayMenu 函数）, 47
displaySale function（displaySale 函数）, 50

displaySizes function（displaySizes 函数）, 86–87
div elements（div 元素）, 312, 319–320, 345, 355
<div> tag（<div>标签）, 307
do keyword（do 关键字）, 22
doAction function（doAction 函数）, 339
DOCTYPE tag（DOCTYPE 标签）, 305
document.getElementById method（document.getElementById 方法）, 310, 316
dot notation（圆点运算符）, 32, 89, 147–151, 160, 166–167, 345
double curly braces（双花括号）, 346
double quotation marks（双引号）, 346

■E

else clause（else 子句）, 200–202, 207
else if statement（else if 语句）, 206, 208–209
empty object（空对象）, 30
equals symbol（等号）, 18
error messages（错误信息）
  checking user input（检查用户输入）, 216–217
  JS Bin（JS Bin）, 9–10
escape sequence（转义符）, 92
exits variable（exits 变量）, 175, 275
exits, in Crypt game（出口，在 The Crypt 游戏中）, 155–163
  adding exits object to full Place constructor（将 exits 对象添加到完整的 Place 构造函数）, 161–163
  creating functions to add and display exits（创建一个添加并显示出口的函数）, 158–159
  giving each place object its own set of exits（设置每个场所对象的出口集合）, 159–161
  testing Place constructor（测试 Place 构造函数）, 163
  using an object to hold exits（使用对象存放出口）, 156–158

■F

failure property（失败属性）, 257, 291
files（文件）
  filenames（文件名）, 225
  importing into other projects（将文件导入其他项目）, 226–229
    adding script element（添加 script 元素）, 227–228
    creating bins（创建 bins）, 227
    refreshing page（刷新页面）, 228
    running program（运行程序）, 228–229
    writing code（编写代码）, 227
  importing multiple files（导入多个文件）, 232–234
  viewing individual code file（查看单个代码文件）, 226
  See also local files（参阅本地文件）
fill function（fill 函数）, 355–356
fillList function（fillList 函数）, 356
findPlanetPosition function（findPlanetPosition 函数）, 71
findTotal function（findTotal 函数）, 46
fitness app（健身应用程序）, 249–253, 265–268
  building（构建）, 374–376
  controllers（控制器）, 281–284
  converting data into user model（将数据转换为用户模型）, 252–253
  creating views（创建视图）, 266–267
  data object（data 对象）, 251–252
  loading data（加载数据）, 368–377
    JS Bin data（JS Bin 数据）, 373–374
    locating user data（加载用户数据）, 368–370
    overview（概述）, 370
    parsing XHR response with JSON.parse（使用 JSON.parse 解析 XHR 响应）, 372–373
    XMLHttpRequest constructor（XMLHttpRequest 构造函数）, 370–372
  user constructor（user 构造函数）, 250–251
  using modules to switch views（使用模块切换视图）, 267–268
floor method（floor 方法）, 205
focus method（focus 方法）, 340
forEach method（forEach 方法）, 110, 113–117, 133, 157
friend variable（friend 变量）, 76–77
fullMessage variable（fullMessage 变量）, 73
function body（函数体）, 44
function invocation operator（函数调用运算符）, 240
function keyword（function 关键字）, 22, 44–45, 56, 106
functions（函数）, 40–56

calling（调用）, 46–48
calling at console prompt（在控制台提示符下进行调用）, 75–76
defining new functions（定义新函数）, 44–45
using function expressions and function declarations（使用函数表达式和函数声明）, 45–46
hiding secret number inside（在函数中隐藏密码数字）, 202–204
hiding variables using（隐藏变量）, 180–181
invoking（调用）, 240–241
passing information to（传递信息）, 59–64
passing multiple arguments to function（将多个实参传递给一个函数）, 63–64
passing one argument to function（将一个实参传递给一个函数）, 59–62
recognizing function expressions（识别函数表达式）, 240
repetition in code and（代码重复）, 40–43
adding tax and displaying summary（加税算出总成本）, 42–43,50–52
displaying object properties as text（将对象的属性作为文本予以显示）, 41–42,48–49
returning data from（返回数据）
return keyword（return 关键字）, 72
return value as replacement for function call（用返回值替代函数调用）, 71
using arguments to determine return value（使用参数来确定返回值）, 72–75
returning objects from（返回对象）, 87–91
reuse and versatility of（函数重用）, 57–59
setting as properties of arguments(methods)(设置为参数的属性（方法）), 91–100
box method（box 方法）, 100
line method（line 方法）, 98–99
math methods（math 方法）, 92–94
namespaces（命名空间）, 91–92
spacer（spacer）, 96–97
string methods（string 方法）, 94–95
wrap method（wrap 方法）, 99–100
using to build objects（构建对象）, 123–127
adding methods（添加方法）, 125–127
adding properties（添加属性）, 124–125
Further Adventures exercises, JS Bin（进阶练习，JS Bin）, 9

■G

game.get() method（game.get()方法）, 336
game.init method（game.init方法）, 296
get method（get方法）, 214,218,287,290–291,334
getBorder function（getBorder函数）, 79–80
getById function（getById函数）, 333
getChallenge method（getChallenge方法）, 258,292
getCount, creating multiple independent counters with（使用getCount创建多个独立计数器）, 181–182
getCounter function（getCounter函数）, 180
getData method（getData方法）, 250–251, 265, 267, 274,276
getExit method（getExit方法）, 191,193,215–217
getExits method（getExits方法）, 139
getExitsInfo function（getExitsInfo函数）, 277
getGame function（getGame函数）, 195–196,214,217,243
getGreeting function（getGreeting函数）, 325
getGuesser function（getGuesser函数）, 202–203
getHelloTo function（getHelloTo函数）, 72–73,77
getInfo method（getInfo方法）, 135–136,272
getItems method（getItems方法）, 359
getItemsInfo function（getItemsInfo函数）, 277
getLastItem method（getLastItem方法）, 191,193,291
getMessage function（getMessage函数）, 71–72, 76
getMessageInfo function（getMessageInfo函数）, 278
getMovieHTML method（getMovieHTML方法）, 313
getPlace method（getPlace方法）, 144,190,193,272,286
getPlanetInfo function（getPlanetInfo函数）, 84,87
getPlayerHealth function（getPlayerHealth函数）, 78
getPlayerInfo function（getPlayerInfo函数）, 79–80, 101,120
getPlayerName function（getPlayerName函数）, 78
getPlayerPlace function（getPlayerPlace函数）, 71,78
getQuestion function（getQuestion函数）, 212
getQuiz function（getQuiz function函数）, 186,210,239
getStateCode function（getStateCode函数）, 149
getTitle method（getTitle方法）, 139
getTitleInfo function（getTitleInfo函数）, 277

getVisitorReport function（getVisitorReport函数）, 109
global namespace（全局命名空间）, 183
global variables（全局变量）, 173, 183, 240
dangers of（危险性）, 172–177
bugs（错误）, 177
naming collisions（命名冲突）, 176
peeking and tweaking（偷窥和篡改）, 173–174
relying on implementation（借助于实现）, 174–176
using window object to check for（使用 window 对象来检查全局变量）, 245–246
go method（go 方法）, 166,214–216,218,287,290, 383–384
checking user input with（检查用户输入）, 215–216
moving player to destination（将玩家移动到目的地）, 216
retrieving player's location（获取玩家位置）, 215
using direction to find destination（通过方向来寻找目的地）, 215
Crypt game（The Crypt 游戏）, 292–293
gpwj namespace（gpwj 命名空间）, 358,373
gpwj.data.load method（gpwj.data.load 方法）, 374
gpwj.templates function（gpwj.templates 函数）, 360
greater than operator（大于运算符）, 209
greater than or equal to operator（大于或等于运算符）, 209
guess function（guess 函数）, 199, 205
guess variable（guess 变量）, 202

■H

<h1> tag（<h1>标签）, 304,307
<h2> tag（<h2>标签）, 307
hasItem method（hasItem 方法）, 270, 272
head section（head 部分）, 319,389
<head> tag（<head>标签）, 308
help resources（获取帮助）, 391–392
HTML（HTML）, 301–322
adding content（添加内容）, 305–306
function declarations（函数声明）, 311
id attribute（id 属性）, 310–311
to web page with JavaScript（使用 JavaScript 向 Web 页面添加内容）, 309–311
when user doesn't have JavaScript（当用户没有 JavaScript 时）, 311
common elements（常见的 HTML 元素）, 307–308
comparing news item data and（对比新闻数据）, 345
constructing by string concatenation（通过连接字符串来构造 HTML）, 345–346
Crypt game（The Crypt 游戏）, 314–322
CSS（CSS 级联样式表）, 319–320
JavaScript's strict mode（JavaScript 严格模式）, 318
message view（message 视图）, 320
modules（模块）, 315–319
placeholders（占位符）, 318–319
playing game（运行游戏）, 320
displaying data from array（从数组显示数据）, 311–314
lists（列表）, 306–307
loading layers（加载图层）, 303–304
overview（概述）, 302–304
starting with empty page（从空白页面开始）, 305
HTML panel, JS Bin（HTML 面板，JS Bin）, 6
html tags（html 标签）, 305

■I

id attribute（id 属性）, 310–311,325,331–332
IDEs (integrated development environments)（IDEs（集成开发环境））, 388
if condition（if 条件）, 293
if keyword（if 关键字）, 22
if statement（if 语句）, 199,351
conditional execution of code（条件执行代码）, 200
using to avoid user input problems（使用 if 语句以避免用户输入问题）, 217–220
if-else blocks（if-else 代码块）, 338
if-else statement（if-else 语句）, 207
IIFE (immediately invoked function expressions)（IIFE（立即调用函数表达式））, 238–242
invoking functions（调用函数）, 240–241

overview（概述），241
recognizing function expressions（识别函数表达式），240
returning information from（返回信息），241–242
implementation（实现），175, 186
importing（导入）
files into other projects（将文件导入其他项目）
adding script element（添加 script 元素），227–228
creating bins（创建 bins），227
refreshing page（刷新页面），228
running program（运行程序），228–229
writing code（编写代码），227
modules（模块），359–360
multiple files（多个文件），232–234
number generator（数字生成器），229–232
picking random questions in quizapp（在测验应用程序中随机选择问题），230–231
using between function in guessing game（在猜数字游戏中使用 between 函数），231–232
index variable（index 变量），117
indexOf method（indexOf 方法），95,98,272, 352
infinite loop（无限循环），352
init method（init 方法），283, 288, 296, 374, 383–384
innerHTML property（innerHTML 属性），310,314, 325
inPlay variable（inPlay 变量），212, 288
input element（input 元素），331
instanceof operator, telling objects apart with（使用 instanceof 运算符来区分对象），131–132
integrated development environments.*See* IDEs interfaces（集成开发环境。见 IDE 界面，174,179–183
creating multiple independent counters with constructor function（用构造函数创建多个独立的计数器），182–183
creating multiple independent counters with getCount（使用 getCount 创建多个独立计数器），181–182
using function to hide variables（使用函数来隐藏变量），180–181
invoking functions（调用函数），46
itemConsumed property（itemConsumed 属性），257,291
itemIndex variable（itemIndex 变量），117
items array（items 数组），113–114,118,120
items parameter（items 参数），137
items variable（items 变量），111

■J

JavaScript panel, JS Bin（JavaScript 面板，JS Bin），7
join array method（join 数组方法），153
join command（join 命令），336–338
join method（jion 方法），110, 130
JS Bin（JS Bin），5–10,223–226
account, getting（获取账户），10
code comments（代码注释），9
creating bins（创建 bins），225
error messages（错误信息），9–10
filenames（文件名），225
following code listings on（在 JS Bin 上运行代码清单），7–8
Further Adventures exercises（进阶练习，JS Bin），9
line numbers（行号），10
loading data（加载数据），373–374
loading HTML layers in（在 JSBin 中加载图层），304
logging to console（输出到控制台），8–9
panels（面板），6–7
Console panel（Console 面板），7
CSS panel（CSS 面板），7
HTML panel（HTML 面板），6
JavaScript panel（JavaScript 面板），7
Output panel（Output 面板），7
specifying Crypt game exits（识别 The Crypt 游戏出口），379–380
viewing individual code file（查看单个代码文件），226
writing code（编写代码），225
JSON.parse（JSON.parse 方法）
converting JSON data into objects and arrays with（将 JSON 转换为对象和数组），378
parsing XHR response with（使用 JSON.parse 解析 XHR 响应），372–373

■K

kalliesCode namespace（kalliesCode 命名空间）, 238
keys array（keys 数组）, 151
key-value pairs, properties as（属性“键-值对”）, 30–32
keywords（关键字）, 22–23

■L

layers, HTML（图层，HTML）, 303–304
length property（length 属性）, 100, 103, 107, 213, 231
less than operator（小于运算符）, 209
less than or equal to comparison operator（比较运算符（小于或等于））, 289
less than or equal to operator（小于或等于运算符）, 209
 tag（<li>标签）, 306,308
line method（line 方法）, 93, 96, 98–99
line numbers, JS Bin（line 号，JS Bin）, 10
lineLength variable（lineLength 变量）, 93
link element（link 元素）, 304, 389
link tag（link 标签）, 303
lists（列表）
  building with template（使用模板构建列表）, 355–356
  HTML（HTML）, 306–307
load function（load 函数）, 373–374, 382
loadData function（loadData 函数）, 370,390
loadNewLevel() method 4（loadNewLevel()方法）
loadPlace method（loadPlace 方法）, 381–382, 384
loadUser method（loadUser 方法）, 369–370,374
local files（本地文件）, 387–391
  concatenating and minifying（合并和压缩）, 390–391
  opening in browser（在浏览器中打开）, 390
  saving（保存）, 388–390
  writing code（编写代码）, 388
local scope（局部范围）, 177
local variables（局部变量）, 240
  assigning namespace properties to（将命名空间属性分配给局部变量）, 246–247
  benefits of（局部变量的优势）, 177–178
log method（log 方法）, 283
longString variable（longString 变量）, 98

■M

map data, Crypt game（地图数据，The Crypt 游戏）, 253–263
  adding challenges to（向 TheCrypt 地图数据添加挑战）, 256–257
  overview（概述）, 255–256
  running game（运行游戏）, 262–263
  updating Place constructor（更新 Place 构造函数）, 258
  using to build game map（建游戏地图）, 258–262
Map Manager module, Crypt game（地图管理器模块，The Crypt 游戏）, 380–384
  adding place to store（添加地点到存储中）, 382
  loading data for single place（为单个地点加载数据）, 382–383
  updating game controller to use（更新游戏控制器）, 383–384
mapData.places array（mapData.places 数组）, 260
math methods（math 方法）, 92–94
Math namespace（Math 命名空间）, 205
Math.ceil method（Math.ceil 方法）, 205
Math.floor（Math.floor 方法）, 326
Math.max method（Math.max 方法）, 93,102
Math.min method（Math.min 方法）, 92
  Math.random() method, generating random numbers with（使用 Math.random()方法生成随机数）, 204–206
Math.round method（Math.round 方法）, 205
message parameter（message 参数）, 60–61
message property（message 属性）, 257,291
message variable（message 变量）, 57–58,60–61
message view（message 视图）
  Crypt game（The Crypt 游戏）, 320
  overview（概述）, 278
  using templates（使用模板）, 362–363

methods（方法）, 91–100
array methods（数组方法）, 110–117
adding and removing elements（添加和删除元素）, 111
forEach method（forEach 方法）, 113–117
slice method（slice 方法）, 112
splice method（splice 方法）, 112–113
box method（box 方法）, 100
line method（line 方法）, 98–99
math methods（math 方法）, 92–94
namespaces and（命名空间）, 91–92
spacer（spacer）, 96–97
string methods（string 方法）, 94–95
wrap method（wrap 方法）, 99–100
minifying local files（压缩本地文件）, 390–391
models（模块）, 248–263
fitness app（健身应用程序）, 249–253
converting data into user model（将数据转换为用户模型）, 252–253
data object（data 对象）, 251–252
user constructor（user 构造函数）, 250–251
map data（地图数据）, 253–263
adding challenges to（向 TheCrypt 地图数据添加挑战）, 256–257
overview（概述）, 255–256
running game（运行游戏）, 262–263
updating Place constructor（更新 Place 构造函数）, 258
using to build game map（构建游戏地图）, 258–262
See also controllers（参见控制器）
modules（模块）, 221–247
collisions（冲突）, 234–238
using namespaces to minimize（使用命名空间来最小化）, 237–238
variable collisions（变量冲突）, 236–237
Crypt game（The Crypt 游戏）
listings（列表）, 317–318
loading（加载）, 318–319
render method（render 方法）, 315–316
immediately invoked function expressions（立即调用函数表达式）, 238–242
invoking functions（调用函数）, 240–241
overview（概述）, 241
recognizing function expressions（识别函数表达式）, 240
returning information from（返回信息）, 241–242
importing files into other projects（将文件导入到其他项目）, 226–229
adding script element（添加 script 元素）, 227–228
creating bins（创建 bins）, 227
refreshing page（刷新页面）, 228
running program（运行程序）, 228–229
writing code（编写代码）, 227
importing multiple files（导入多个文件）, 232–234
importing number generator（导入数字生成器）, 229–232
picking random questions in quiz app（在测验应用程序中随机选择问题）, 230–231
using between function in guessing game（在猜数字游戏中使用 between 函数）, 231–232
JS Bin（JS Bin）, 223–226
creating bins（创建 bins）, 225
filenames（文件名）, 225
viewing individual code file（查看单个代码文件）, 226
writing code（编写代码）, 225
news page (fitness app)（健身应用程序）, 357–360
importing modules（导入模块）, 359–360
news data（新闻数据）, 358–359
templating functions（模板函数）, 357–358
organizing code into（组织代码）, 242–247
sharing namespace across modules（跨模块共享命名空间）, 244–247
using to switch views（使用模块切换视图）, 267–268
moons property（moons 属性）, 129
mountain variable（mountain 变量）, 172–173,177
move function（move 函数）, 89–90
movie variable（movie 变量）, 49
multiplication operator（乘法运算符）, 21
My Movie Ratings page（My Movie Ratings 页面）, 325,329

■N

name parameter（name 参数）, 76,250
namespaces（命名空间）, 91–92, 183
minimizing collisions with（减少冲突）, 237–238
sharing across modules（跨模块共享）, 244–247
assigning namespace properties to local variables（将命名空间属性分配给局部变量）, 246–247
window object（window对象）, 244–246
using object as（将对象用作）, 184–185
name-value pair（名称-值对）, 31
new keyword（new 关键字）, 127, 157, 182, 188, 270
new-line character（换行符）, 91–92
newLine function（newLine 函数）, 91–92
news page (fitness app)（新的页面(健身应用程序)）, 344–347, 357–360
comparing news item data and HTML（对比新闻数据和HTML）, 345
constructing HTML by string concatenation（通过连接字符串来构造HTML）, 345–346
designing with HTML templates（使用HTML模板进行设计）, 346
modules（模块）, 357–360
importing（导入）, 359–360
news data（新闻数据）, 358–359
templating functions（模板函数）, 357–358
using script tags for templates（使用script标签包裹模板）, 346–347
next function（next 函数）, 183, 214
nextYear array（nextYear 数组）, 106
not strictly equal to operator（不严格相等运算符）, 209
notes property（notes 属性）, 36
null value（空值）, 141
Number Generator code（数字生成器代码）, 223
number generator, importing（导入数字生成器）, 229–232
picking random questions in quiz app（在测验应用程序中随机选择问题）, 230–231
using between function in guessing game（在猜数字游戏中使用between函数）, 231–232
numberOfHours variable（numberOfHours变量）, 21
numberToSquare parameter（numberToSquare参数）, 62

■O

object properties, matching template placeholders with（将模板占位符与对象属性相匹配）, 353–354
Object.keys method（Object.keys方法）, 151,354
objects（对象）, 27–39, 83–103
accessing properties of（访问属性）, 32–33
converting JSON data into objects and arrays with（将JSON转换为对象和数组）, 378
creating（创建）, 29–32
empty object（空对象）, 30
properties as key-value pairs（属性“键-值对”）, 30–32
Crypt game player object（The Crypt游戏的玩家对象）, 37–39
overview（概述）, 28–29
player objects as arguments（玩家对象作为参数）, 101–103
returning from functions（从函数返回）, 87–91
setting functions as properties of(methods)（设置函数作为属性（方法））, 91–100
box method（box方法）, 100
line method（line方法）, 98–99
math methods（math方法）, 92–94
namespaces（命名空间）, 91–92
spacer（spacer）, 96–97
string methods（string方法）, 94–95
wrap method（wrap方法）, 99–100
updating properties of（更新属性）, 33–35
using as arguments（使用对象作为参数）, 84–87
accessing properties of object argument（访问对象参数的属性）, 84–85
adding properties to object argument（增加对象参数的属性）, 85–87

using as namespace（用作命名空间），184–185
using constructors to build（使用构造函数来构建），127,132
overview（概述），127,130
telling objects apart with instanceof operator（使用instanceof运算符来区分对象），131–132
world building-making use of Planet constructor（使用Planet构造函数创建一个新世界），130–131
using functions to build（使用函数来构建），123–127
adding methods（添加方法），125–127
adding properties（添加属性），124–125
 tag（<ol>标签），306–308
opening tags（开始标签），304
option elements（option元素），327–328
options array（options数组），133
OR operator（OR运算符），293
Output panel, JS Bin（输出面板，JS Bin），7

■P

<p> tag（<p>标签），304, 306–307
parameters vs. arguments（形参与实参），62
parentheses（圆括号），44, 47
parseCommand function（parseCommand函数），339
peakCount function（peakCount函数），182
Place constructor, updating（Place构造函数，更新），258
Place object（Place对象），135, 139, 144,158,161,166
place property（place属性），142, 144
place variable（place变量），215–216
place views（place视图），274–277
model（模型），274–276
using templates（使用模板），364–365
view（视图），276–277
placeholders（占位符）
automating placeholder replacement for templates（自动替换模板中的占位符），353–356
building lists（构建列表），355–356
filling all placeholders for each key（为每个“键”填充所有的占位符），355
matching template placeholders with object properties（将模板占位符与对象属性相匹配），353–354
Crypt game（The Crypt游戏），318–319
See also templates（另见模板）
places array（places数组），255
placesStore object（placesStore对象），260
placeView function（placeView函数），269
Planet function（Planet函数），127,129
planet parameter（planet参数），84,87
planet1 variable（planet1变量），86
planet.area property（planet.area属性），87
planets variable（planets变量），126–127
planet.volume property（planet.volume属性），87
player commands, Crypt game（玩家命令，The Crypt游戏），334–342
adding controls to page（向页面添加控件），335–336
Commands module（命令模块），340–342
join command（jion命令），336–338
listening for button clicks（监听按钮的点击），339–340
mapping text box entries to game commands（将文本框内容映射到游戏命令），336
pop command（pop命令），336–338
shift command（shift命令），336–338
split command（split命令），336–338
switch block（switch结构），338–339
Player constructor（Player构造函数），243, 245, 270–271,285
player information, displaying（玩家信息，显示），64–69
health（健康值），66–67
locations（位置），67–68
names（名字），65–66
Player module（Player模块），245
Player objects（Player对象），140, 165
player variable（player变量），54, 56, 64, 288
player views（player视图），269–274
model（模型），270–272
using templates（使用模板），363–364

view（视图）, 272–274
playerHealth parameter（playerHealth参数）, 66
playerInfo variable（playerInfo变量）, 79
playerName parameter（playerName参数）, 66–67
playerPlace parameter（playerPlace参数）, 67
player.place.items array（player.place.items数组）, 166
players array（players数组）, 115–116
playerView function（playerView函数）, 269
polluting global namespace（污染全局命名空间）, 183, 237
pop command, Crypt game（pop命令，The Crypt游戏）, 336–338
pop method（pop方法）, 110–111
posted key（posted“键”）, 354
posted property（posted属性）, 349
programming（编程）, 3–4
properties of objects（对象的属性）
accessing（访问）, 32–33
as key-value pairs（键-值对）, 30–32
updating（更新）, 33–35
properties, displaying as text（属性，显示为文本）, 41–42
property values（属性值）, 33
prototypes（原型）, 123
push method（push方法）, 110–111,130

■Q

qIndex property（qIndex属性）, 184–186
qIndex variable（qIndex变量）, 212–213
questions array（questions数组）, 212–213, 231
quiz app（测验应用程序）, 37,210–214
checking player’s answer（检测用户答案）, 213
creating（创建）, 183–187
hiding questions array（隐藏questions数组）, 185–187
using object as namespace（将对象用作命名空间）, 184–185
displaying question（显示问题）, 212–213
handling player’s answer（处理用户的答案）, 214
moving to next question（显示下一个问题）, 213
multiple declarations with single var keyword（使用var关键字声明多个变量）, 211–212
picking random questions in（挑选随机问题）, 230–231
returning interface object（返回接口对象）, 214
quiz variables（quiz变量）, 183,185,239
quizMe method（quizMe方法）, 183,186,210,230
quiz.next（quiz.next）, 185
QuizQuestion constructor（QuizQuestion构造函数）, 132
quiz.questions array（quiz.questions数组）, 184
quiz.showMe（quiz.showMe）, 185
quiz.submit() method（quiz.submit()方法）, 214

■R

random numbers, generating with Math.random() method（随机数，使用Math.random()方法生成）, 204–206
Rate button（Rate按钮）, 327
rateMovie function（rateMovie函数）, 329–330,332–334
refreshing page, JS Bin（刷新页面，JS Bin）, 228
regular expressions, replacing string multipletimes with（正则表达式，用多次替换字符串）, 353
removeItem method（removeItem方法）, 270, 272
render method（render方法）, 272, 277–278, 313–317,362
renderMessage（renderMessage）, 289, 390
repetition in code（代码重复）, 40–43
adding tax and displaying summary（加税算出总成本）, 42–43
displaying object properties as text（将对象的属性作为文本予以显示）, 41–42
replace method（replace方法）, 347–348, 350
requires property（requires属性）, 257,291
reserved words, for naming variables（保留字，变量命名）, 22–23
resources（资源）, 391–392
responseText property（responseText属性）, 372
result variable（result变量）, 46
return keyword（return关键字）, 72,83
return statement（return语句）, 72, 129

return value（返回值）
as replacement for function call（替代函数调用）, 71
determining using arguments（使用参数确定）, 72–75
returning data from functions（从函数返回数据）, 70–75
return keyword（return关键字）, 72
return value as replacement for function call（用返回值替代函数调用）, 71
using arguments to determine return value（使用参数来确定返回值）, 72–75
running program, JS Bin（运行程序，JS Bin）, 228–229

■S

sale variable（sale变量）, 52
saving local files（保存本地文件）, 388–390
Say Hi button（Say Hi按钮）, 324
sayHello function（sayHello函数）, 44, 46
scope（作用域）, 171–197
Crypt game（The Crypt游戏）
hiding place info（隐藏地点信息）, 191–193
hiding player info（隐藏玩家信息）, 187–191
user interaction（用户交互）, 193–197
global variables, dangers of（全局变量，危险性）, 172–177
bugs（错误）, 177
naming collisions（命名冲突）, 176
peeking and tweaking（偷窥和篡改）, 173–174
relying on implementation（借助于实现）, 174–176
interfaces（接口）, 179–183
creating multiple independent counters with constructor function（用构造函数创建多个独立的计数器）, 182–183
creating multiple independent counters with getCount（使用getCount创建多个独立计数器）, 181–182,
using function to hide variables（使用函数来隐藏变量）, 180–181
local variables, benefits of（局部变量，优势）, 177–178
quiz app, creating（创建测验应用程序）, 183–187
hiding questions array（隐藏questions数组）, 185–187
using object as namespace（将对象用作命名空间）, 184–185
score property（scores属性）, 34–35
score variable（score变量）, 17
script element, adding（script元素，添加）, 227–228
script tags, using for templates（script标签，用于模板）, 346–347
secret variable（secret变量）, 202
secretMountain variable（secretMountain变量）, 178
select element（select元素）, 327–330
adding to page（添加到页面）, 328–329
rateMovie function（rateMovie函数）, 329–330
selected attribute（selected属性）, 329
self-closing tag（自动闭合标签）, 331
semicolon character（分号字符）, 10
sessions array（sessions数组）, 250
setPlace method（setPlace方法）, 190, 193, 215, 272
shift array（shift数组）, 337
shift command, The Crypt game（shift命令，The Crypt游戏）, 336–338
show function（show函数）, 178
showArguments function（showArguments函数）, 115–116
showCostBreakdown function（showCostBreakdown函数）, 44
showEvent method（showEvent方法）, 133–134
showExits function（showExits函数）, 158–159
showInfo method（showInfo方法）, 113–114,165,190,217
showMe function（showMe函数）, 183, 186
showMessage function（showMessage函数）, 57–58, 60–62
showMountain function（showMountain函数）, 173
showMovieInfo function（showMovieInfo函数）, 44, 49, 55
showPlanet method（showPlanet方法）, 125–126
showPlayerHealth function（showPlayerHealth函数）, 68
showPlayerInfo function（showPlayerInfo函数）, 55, 64, 68,84,120

showPlayerName function（showPlayerName函数）, 65,68
showPlayerPlace function（showPlayerPlace函数）, 67–68
showQuestion function（showQuestion函数）, 133
showSum function（showSum函数）, 63, 70
slice method（slice方法）, 110,112
smooth operator（平稳的运算符）, 18
social property（social属性）, 349
spacer function（spacer函数）, 236
spacer.line（spacer.line方法）, 97
spacer.newLine method（spacer.newLine方法）, 139
spacer.wrap（spacer.wrap方法）, 97
span element（span元素）, 333,345
splice method（splice方法）, 110,112–113
split command, The Crypt game（shift命令，The Crypt游戏）, 336–338
split method（split方法）, 153–154
square brackets（方括号）, 107,149
square function（square函数）, 62
src attribute（src属性）, 228–229,297,390
statement（语句）, 10
states object（states对象）, 148–149
strict equality operator（严格相等运算符）, 199–200
strict mode (JavaScript) （严格模式(JavaScript)）, 318
strictly equal to operator（表示严格相等的运算符）, 209
string concatenation operator（字符串连接运算符）, 20
string methods（string方法）, 94–95
strings（字符串）
constructing HTML by string concatenation（通过连接字符串来构造HTML）, 345–346
replacing multiple times（多次替换）, 349–353
repeating code while condition is met（在满足条件时重复执行代码）, 350–351
replacing string while it can be found（替换一个可以找到的字符串）, 352–353
replacing strings with regular expressions（使用正则表达式替换字符串）, 353
while loop（while循环）, 351–352
replacing one string with another（用一个字符串替换另一个）, 347–349
style tag（style标签）, 303
submit method（submit方法）, 210
substr method（substr方法）, 94–95
success property（success属性）, 257,291
switch block, The Crypt game（switch结构，The Crypt游戏）, 338–339
switch keyword（switch关键字）, 22
switch variable（switch变量）, 338

■T

template.fill method（template.fill方法）, 362
templates（模板）, 37,343–366
automating placeholder replacement for（自动替换占位符）, 353–356
building lists（构建列表）, 355–356
filling all placeholders for each key（为每个“键”填充所有的占位符）, 355
matching template placeholders with object properties（将模板占位符与对象属性相匹配）, 353–354
news page (fitness app)（新页面（健身应用程序））, 344–347, 357–360
comparing news item data and HTML（对比新闻数据和HTML）, 345
constructing HTML by string concatenation（通过连接字符串来构造HTML）, 345–346
designing with HTML templates（使用HTML模板进行设计）, 346
modules（模块）, 357–360
using script tags for templates（使用script标签包裹模板）, 346–347
replacing strings（替换字符串）
multiple times（多次）, 349–353
replacing one string with another（用一个字符串替换另一个）, 347–349
views（视图）, 360–366
creating HTML templates for（创建HTML模板）, 361–362
updating to use new templates（针对新模板更新视图）, 362–365
Templates module（模板模块）, 361

text boxes（文本框）, 330–334
adding to page（添加到页面）, 331
adding unordered list to display comments（添加无序列表以显示注释）, 332
CSS（级联样式表）, 334
getting references to new elements（获取对新元素的引用）, 332
rateMovie function（rateMovie函数）, 332–334
text, displaying object properties as（将对象的属性作为文本进行显示）, 41–42
this variable（this变量）, 157,172, 182–183,187–188, 190–191,193
thisYear array（thisYear数组）, 106
tightly coupled（紧密）, 350
title element（title元素）, 318–319
title key（title键）, 354
title property, 135, 157（title属性）, 349
total property（total属性）, 251
total variable（total变量）, 74
totalCost function（totalCost函数）, 74–75
toUpperCase method（toUpperCase方法）, 87, 94, 96
type attribute（type属性）, 331,346
type property（type属性）, 336

■U

 tag（<ul>标记）, 306–308, 313
unordered lists, adding to display comments（无序列表，用于显示评论内容）, 332
unshift method（unshift方法）, 130
updateGreeting function（updateGreeting函数）, 325–326
updating properties of objects（更新对象属性）, 33–35
use function（use函数）, 287, 290, 294–296, 334
use strict statement（use strict语句）, 326
user constructor, fitness app（user构造函数，健身应用程序）, 250–251
user input, checking（用户输入，检查）, 214–220
error messages（错误信息）, 216–217
go method（go方法）, 215–216
moving player to destination（将玩家移动到目的地）, 216
retrieving player's location（获取玩家的位置）, 215
using direction to find destination（通过方向来寻找目的地）, 215
using if statement to avoid problems（使用if语句避免问题的出现）, 217–220
user model, converting data into（将数据转换为用户模型）, 252–253
userView variable（userView变量）, 266

■V

value property（value属性）, 330, 332, 340
values, assigning to variables（为变量赋值）, 18–19
one-step declaration and assignment（一步实现变量声明和赋值）, 20
using variable in its own assignment(先运算再赋值), 21–22
var keyword（var关键字）, 17, 22, 28, 30, 35, 211–212
variable collisions（变量冲突）, 236–237
variables（变量）, 16–26
assigning namespace properties to local variables（将命名空间属性分配给局部变量）, 246–247
assigning values to（赋值）, 18–19
one-step declaration and assignment（一步实现变量声明和赋值）, 20
using variable in its own assignment（先运算再赋值）, 21–22
choosing good names for（选择合适的变量名）, 22–24
camel casing（骆驼命名法）, 23
descriptive names（描述性变量名）, 24
keywords and reserved words（关键字和保留字）, 22–23
rules for naming variables（变量的命名规则）, 23
crypt game player variables（TheCrypt游戏的玩家变量）, 24–26
declaring（声明）, 17–18, 20
declaring at console prompt（在控制台提示符处声明）, 76–77
global, dangers of（危险的全局变量）, 172–177
bugs（漏洞）, 177
naming collisions（命名冲突）, 176

peeking and tweaking（偷窥和篡改), 173–174
relying on implementation（借助于实现）, 174–176
local, benefits of（局部变量的优势), 177–178
overview（概述), 17
using window object to check for global variables(使用window对象来检查全局变量), 245–246
views（视图), 264–279, 360–366
creating HTML templates for（创建HTML模板), 361–362
fitness app（健身应用程序), 265–268
creating views（创建视图), 266–267
using modules to switch views（使用模块切换视图), 267–268
message view（消息视图), 278
place views（场所视图), 274–277
model（模型）, 274–276
view（视图), 276–277
player views（玩家视图), 269–274
model（模型), 270–272
view（视图), 272–274
updating to use new templates（针对新模板更新视图), 362–365
messages（消息), 362–363
places（场所), 364–365
players（玩家), 363–364
See also controllers（另见控制器）
visitorArray variable（visitorArray变量), 109
visitors variable（visitors变量), 109
volume property（volume 属性), 85

■W

weather app（天气应用程序), 36–37
web page, adding content with JavaScript（使用JavaScript向Web页面添加内容), 309–311
function declarations（函数声明), 311
id attribute（id属性), 310–311
when user doesn't have JavaScript（当用户没有JavaScript 时), 311
while loops, replacing string multiple times（While循环——多次替换字符串), 349–353
overview（概述), 351–352
repeating code while condition is met（在满足条件时重复执行代码), 350–351
replacing string while it can be found（替换一个可以找到的字符串), 352–353
with regular expressions（使用正则表达式), 353
window object, （window对象） 244–246
word counts（字数统计), 152–155
words objec（words对象）t, 152
wrap method（wrap方法), 9–100
wrap-and-return process（打包-返回过程), 196
wrapping code（打包代码), 196
writing blog（撰写博客), 35
writing code（编写代码）
JS Bin（JS Bin), 227
local files（本地文件), 388

■X

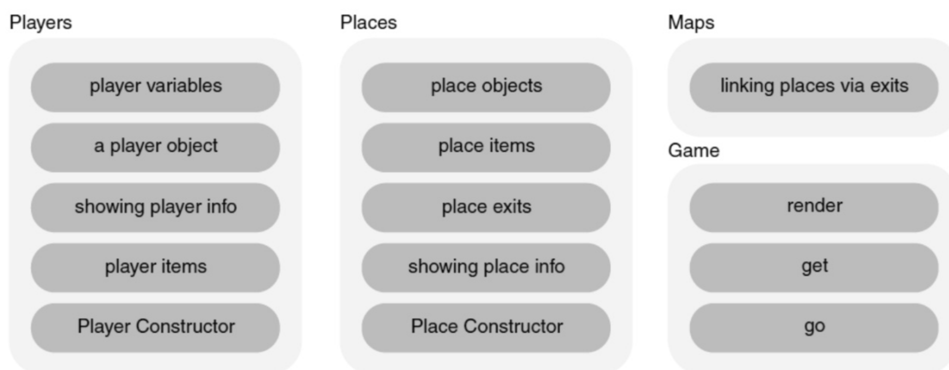
XHR data loading（XHR数据加载), 367–386
Crypt game（TheCrypt游戏), 378–386
building game page（构建游戏页面), 384–385
cache（缓存), 380
Map Manager module（地图管理器模块）, 380–384
specifying exits with JS Bin file codes（使用JSBin文件代码指定出口), 379–380
fitness app（健身应用程序), 368–377
building（升级), 374–376
JS Bin data（JS Bin 数据), 373–374
locating user data（查找用户数据), 368–370
overview（概述), 370
parsing XHR response with JSON.parse（使用JSON.parse解析XHR响应), 372–373
XMLHttpRequest constructor（XMLHttpRequest构造函数), 370–372
JSON data（JSON 数据), 377–378
XMLHttpRequest object（XMLHttpRequest对象), 367, 369–372, 378

■Y

yield keyword（yield 关键字), 22

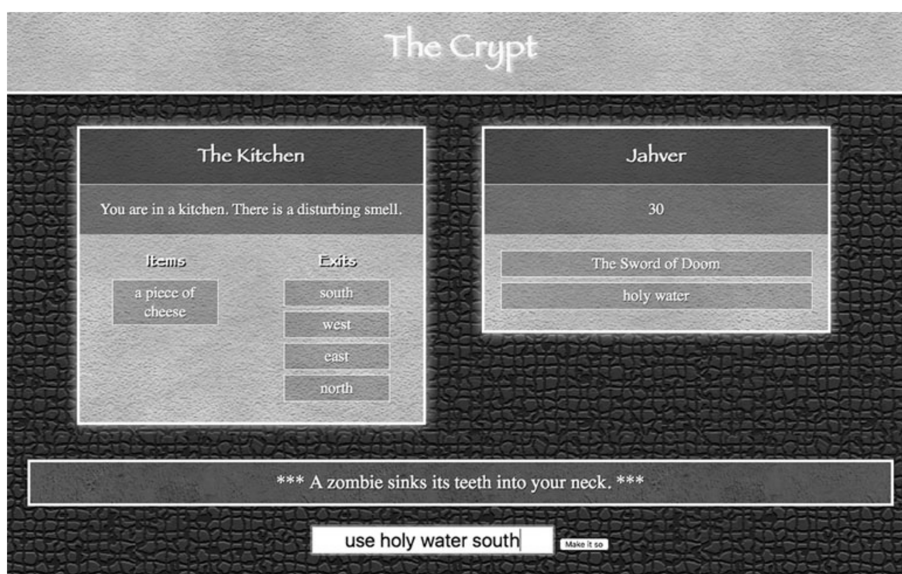
The Crypt：一个持续示例

在本书的第一部分，读者在学习 JavaScript 核心概念的同时，就可以编写代码来表示游戏中的玩家和地点，而且可以让玩家在各个场所之间来回移动并拾取物品。下图展示了该游戏中的基本构件。在本书的每一章都有一张类似的图，展示了在整体游戏中该章所涉及的部分内容。



在本书的第二部分，对玩家增加了挑战：在没有解决谜题之前禁止退出。这部分的重点是如何组织代码，如何隐藏代码的实现方式，如何检查用户输入，如何构建可以重复使用和替换的模块来提高项目的灵活性。

在本书的第三部分，读者可以使用 HTML 模板来更新游戏的显示界面，进一步完善游戏,使其在运行过程中就可以加载数据，还可以使用玩家和场所信息来填充模板，并添加文本框和按钮，以便玩家可以通过网页输入命令。



在线互动交流平台
官方微博：http://weibo.com/cmpjsj
豆 瓣 网：http://site.douban.com/139085/
读者信箱：cmp_itbook@163.com



JavaScript开发实战

Get Programming with JavaScript

计算机科学先进技术译丛

- 实用机器学习
- AKKA实战
- JavaScript开发实战
- 敏捷开发实战
- GO实战
- Spark实战

内容简介

本书是一本JavaScript开发入门指导书，主要介绍了变量、对象、函数、数组、构造函数、方括号运算符、作用域、条件语句、模块、模型、视图、控制器、构建网页、控件、模板、数据加载等内容。书中涉及的概念都附有简短示例。此外，还有一个贯穿全书的持续示例——冒险游戏The Crypt。读者可以在一个交互式网站上直接运行书中的代码。


机械工业出版社计算机分社官方微信

关注计算机分社官方微信，回复您购买图书书号中间的五位数字，即可获取本书配套资源下载链接，并可获得更多增值服务和最新资讯。

ISBN 978-7-111-58671-5

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
服务咨询热线：010-88361066
读者购书热线：010-68326294
010-88379203
网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
金 书 网：www.golden-book.com
教育服务网：www.cmpedu.com
封面无防伪标均为盗版


机械工业出版社微信公众号


机工IT互联网工厂微信服务号

上架指导 程序设计

ISBN 978-7-111-58671-5

策划编辑◎ 车忱 / 封面设计◎  子时文化 ZiShi Culture



定价：89.00元